CONCORRENZA - INTRODUZIONE
COME PROGETTARE E REALIZZARE SISTEMI CONCORRENTI CORRETTI?
CORRETTEZZA DI PROGRAMMI SEQ
RIPASSO: LOGICA PROPOSIZIONALE
LOGICA PROPOSIZIONALE - SINTASSI
LOGICA PROPOSIZIONALE - SEMANTICA
MODELLI
LOGICA PROPOSIZIONALE - APPARATO DEDUTTIVO
LOGICA DI HOARE
PRIMO LIVELLO
SECONDO LIVELLO
PSEUDO LINGUAGGIO USATO
LOGICA DI HOARE - APPARATO DEDUTTIVO
PROPRIETÀ GENERALI DELLA LOGICA DI HOARE
ALCUNE NOZIONI
OSSERVAZIONI
LOGICA PROPOSIZIONALE E INSIEMI DI STATI
CRITERIO DI SCELTA DELLA PRECONDIZIONE MIGLIORE
REGOLE DI CALCOLO DI WP
CONTESTI DIVERSI
DESIGN BY CONTRACT
STORIA
DIMOSTRAZIONI DI CORRETTEZZA
LOGICA DI HOARE E SINTESI DI PROGRAMMI ITERATIVI
SCHEMA GENERALE DI DIMOSTRAZIONE
ESEMPI DI DERIVAZIONE
ASSEGNAMENTO
ASSEGNAMENTO
CONTROESEMPI
SEQUENZA
SCELTA
ITERAZIONE - ESPONENZIALE
ITERAZIONE - DIVISIONE
CORRETTEZZA TOTALE - ESEMPIO
CORRETTEZZA TOTALE - VARIABILE NON DECREMENTA
CORRETTEZZA TOTALE - ESPONENZIALE
CORRETTEZZA TOTALE - FATTORIALE
CORRETTEZZA TOTALE - ESEMPIO COMPLESSO: MOLTIPLICAZIONE RUSSA
CORRETTEZZA TOTALE - QUADRATO
MODELLI DI SISTEMI CONCORRENTI
CALCULUS COMMUNICATING SYSTEMS
LABELLED TRANSITION SYSTEMS (LTS)
NOTAZIONI
CCS PURO
OPERATORI CCS
SPECIFICHE E IMPLEMENTAZIONI
EQUIVALENZA RISPETTO ALLE TRACCE
BISIMULAZIONE FORTE
ESEMPI
BISIMULAZIONE DEBOLE
TRANSIZIONE DEBOLE
BISIMULAZIONE DEBOLE: DEFINIZIONE
PROPRIETÀ
PROCESSI DETERMINISTICI ED EQUIVALENZE
GIOCO ATTACCANTE E DIFENSORE

ESEMPI

CONCORRENZA - INTRODUZIONE

Concorrere singifica correre insieme; i sistemi concorrenti quindi rappresentano diversi componenti che operano in contemporanea. In particolare questi componenti possono coordinarsi per un obiettivo comune ma anche competere per le risorse disponibili.

Possiamo trovare la concorrenza ovunque, alcuni esempi di sistemi concorrenti sono:

- **Cellula vivente**: all'interno di essa avvengono in contemporanea molti processi come l'assemblaggio di proteine, la duplicazione del dna, ecc. Nella cellula non c'è un *direttore*, dunque è un sistema ***asincrono***.
- **Orchestra musicale**: in questo sistema, a differenza della cellula, c'è un direttore che da il tempo agli altri componenti; è un sistema ***sincrono***, ovvero che fa riferimento ad un *cronometro* condiviso da tutti i componenti e dove ogni componente sa che deve eseguire una certa azione in un certo momento.
- **Processore multicore**: un processore dove possono essere eseguite diverse operazioni in contemporanea.
- **Squadra antincendio**: sistema dove i componenti hanno un obiettivo comune: spegnere l'incendio e si coordinano a tal scopo.

COME PROGETTARE E REALIZZARE SISTEMI CONCORRENTI CORRETTI?

Per realizzare dei sistemi concorrenti corretti abbiamo bisogno di:

- **Linguaggi**: ovviamente per poter realizzare dei programmi concorrenti abbiamo bisogno di linguaggi di programmazione che permettano di stabilire quali operazioni vanno eseguite in parallelo e quali in serie.

  Un esempio pratico è Java con la classe *Thread*.
  Un esempio più astratto è la ***partitura musicale*** che rappresenta le varie parti che un'orchestra deve suonare. Essendo l'orchestra un sistema sincrono inoltre, questo si riflette anche nel linguaggio; nelle partiture il tempo è indicato allo stesso modo per tutte le parti.

  Un linguaggio astratto può essere rappresentato dai sistemi di equazioni *(ad esempio **CCS** che vedremo in seguito)*, che permettono di scrivere dei programmi estremamente astratti concentrandoci sulla comunicazione fra le parti più che al trattamento dei dati.

- **Modelli**: avremmo bisogno anche di modelli, ovvero di modi per rappresentare il sistema in una maniera semplificata rispetto al sistema reale mantenendo però le caratteristiche cruciali della concorrenza. Vogliamo analizzare questi modelli anziché i programmi concorrenti poiché sono più semplici e non presentano dettagli presenti invece nei programmi che non sono necessariamente collegati alla concorrenza, permettendoci così di concentrarci solo sugli aspetti effettivamente rilevanti.

  Un esempio di modello sono i *grafi di attività*, ovvero un grafo dove i nodi rappresentano delle attività e gli archi rappresentano la precedenza tra le varie attività; i nodi non ordinati fra loro possono essere eseguiti parallelamente.
  Altri modelli sono ad esempio le reti di Petri. Saranno studiati in seguito.

- **Logica**: per esprimere i criteri secondo i quali un certo sistema è corretto useremo la logica.

CORRETTEZZA DI PROGRAMMI SEQ

Consideriamo il seguente programma sequenziale

```
int f (int v[], int n) {
    int x = v[0]; int p = 0; int a = 1;
    while(a < n) {
        if (v[a] < x) {
            x = v[a]; p = a;
        }
        a = a + 1;
    }
    return p;
}
```

SI tratta sostanzialmente di una *funzione* f che prende in input un vettore di interi v e un intero n rappresentante la dimensione del vettore v e calcola il valore minimo contenuto in esso. Infine, restituisce la posizione del valore minimo calcolato.

Come possiamo dimostrare la sua correttezza?
Prima di poter dire come si può dimostrare che il programma sia corretto è necessario definire più esattamente possibile cosa vuol dire che un programma sia *corretto*.

DEFINIAMO IL CRITERIO DI *CORRETTEZZA*
Per definire un criterio che sia *rigoroso*, appelliamo alla matematica e al suo linguaggio *preciso*.
Per quanto riguarda il programma di esempio, vorremmo specificare la richiesta che ritorni sempre la posizione dell'elemento minimo; possiamo farlo utilizzando una formula di questo tipo:

Alla fine dell'esecuzione $\forall i \in \{0, \ldots, n-1\} \quad v[p] \leq v[i]$ *(postcondizione)*
Ovvero, vogliamo che alla fine dell'esecuzione il valore indicato da p sia minore o uguale di tutti gli altri valori del vettore.
In genere però non basta specificare cosa si vuole che sia vero alla fine dell'esecuzione; in molti casi un programma viene scritto per soddisfare una condizione alla fine dell'esecuzione purché valga qualcosa all'inizio dell'esecuzione. Nel nostro esempio questo è banale e può essere specificato come segue:

Prima dell'esecuzione $\forall i \in \{0, \ldots, n-1\} \quad v[i] \in \mathbb{Z}$ *(precondizione)*
Ovvero, vogliamo che prima dell'esecuzione i valori del vettore v siano dei numeri interi, visto che era stato dichiarato come vettore di interi.
Abbiamo specificato cosa intendiamo dire quando diciamo una funzione è corretta; diremo che la funzione f è *soddisfacente* o *corretta* se ogni volta che i dati, prima dell'esecuzione, soddisfano la precondizione alla fine dell'esecuzione soddisferanno la postcondizione.

DOMANDA: come mai esistono programmi in grado di decidere la correttezza sintattica di un programma *(i compilatori)*, ma non esistono programmi che, dati in input precondizioni, postcondizioni e codice del programma siano in grado di deciderne la correttezza semantica?

Perché è un problema indecidibile. Basti pensare all'*halting problem*; un programma corretto è un programma che sicuramente termina, ma non possiamo decidere se termina o meno.
Tuttavia questo significa che non possiamo creare un verificatore generale di correttezza, ma non che non sia possibile decidere se un singolo programma sia o meno corretto.

RIPASSO: LOGICA PROPOSIZIONALE

Prima di parlare di logica proposizionale è necessario inquadrare cosa si intende più in generale con logica *formale* o *matematica*; quando si parla di logica in matematica si intende un **sistema** formato da un ***linguaggio***, e quindi da una grammatica che specifica come costruire delle frasi *(formule)*.

Possiamo quindi costruire delle formule e lo facciamo con lo scopo di assegnare loro un significato; quindi, fissata una logica come linguaggio, vogliamo definire una ***semantica***, ovvero un modo per attribuire un significato agli elementi del linguaggio e in particolare alle formule.
Nel caso della logica proposizionale, dal punto di vista teorico, il significato di una formula è il suo **valore di verità**.

Una volta definite sintassi e semantica, possiamo usare una logica per costruire delle ***dimostrazioni***; aver definito una semantica non vuol dire che di fronte ad una formula si è in grado di dire subito se sia valida o meno, dunque con la logica si sviluppa un metodo di dimostrazione e si cerca, in un numero finito di passi, di dimostrare la validità di una certa formula.
Un esempio, legato alla logica dell'aritmetica, è la formula che dice che per ogni numero primo x, esiste sempre un numero primo y più grande di esso; questa formula non è verificabile sperimentalmente essendo in numeri infiniti e va quindi dimostrata in altri modi.

Per ogni logica bisogna quindi definire un **apparato deduttivo**, cioè un insieme di ***regole di inferenza***.
Una regola di inferenza è una regola che dice *"se hai già dimostrato queste premesse, allora puoi dedurre questa formula"*.

LOGICA PROPOSIZIONALE - SINTASSI

Passiamo ora al caso specifico della logica proposizionale, considerata come la logica più semplice e che ci servirà per le dimostrazioni di correttezza.
Per costruire il linguaggio avremmo bisogno di

- $PA = \{p_1, \ \dots, \ p_i, \ \dots\}$ Proposizioni atomiche *(frasi, espressioni aritmetiche, ecc)*
- $\bot$, T Costanti logiche, sono formule sempre false/vere
- $\neg$, $\vee$, $\wedge$, $\rightarrow$, $\leftrightarrow$ Connettivi logici, utili a creare formule complesse
- (,) Delimitatori, sempre per le formule complesse

Adesso dobbiamo definire la ***grammatica*** della logica proposizionale e lo faremo induttivamente.
Definiamo F_p insieme delle **formule ben formate**:

1. $\bot, \mathsf{T} \in F_p$ Le costanti logiche sono fbf
2. $\forall p_i \in PA, pi \in F_p$ Le proposizioni atomiche sono fbf
3. Se $A, B \in F_p$, allora $(\neg A)$, $(A \vee B)$, $(A \wedge B)$, $(A \rightarrow B)$, $(A \leftrightarrow B) \in F_p$

Esempio: $(x < 0 \wedge (\neg(y \geq z)))$ *(in questo caso $x < 0 \in PA$)*

LOGICA PROPOSIZIONALE - SEMANTICA

Adesso vogliamo attribuire un significato alle formule definite prima.
Siamo interessati a sapere se una formula sia vera o meno. Il valore di verità di una formula tuttavia dipende dai valori di verità delle sue proposizioni atomiche.

Il punto di partenza è quindi stabilire quali proposizioni atomiche considerare vere: questo si può fare formalmente definendo una

Funzione di valutazione $v : PA \to \{0,\ 1\}$ *(0 e 1 stanno per False e True)*

Estendiamo ora v induttivamente ad una funzione definita su tutte le formule ben formate
$I_v : F_p \to \{0,1\}$ *funzione di interpretazione*

1. $I_v(\bot) = 0$ $I_v(\mathsf{T}) = 1$ Falso e vero hanno sempre valori 0/1
2. $\forall p_i \in PA$ $I_v(p_i) = v(p_i)$ L'interpretazione riprende il valore di v per le PA
3. $I_v(\neg A) = 1 - I_v(A)$ Passo induttivo
   $I_v(A \vee B) = 1$ sse $I_v(A) = 1$ o $I_v(B) = 1)$
   ...
   $I_v(A \to B) = 1$ sse $I_v(B) = 1$ o $I_v(A) = 0$

Qualche termine

- A è SODDISFATTA da I_v se $I_v(A) = 1$
- A è SODDISFACIBILE se esiste I_v tale che $I_v(A) = 1$
- A è una TAUTOLOGIA se $I_v(A) = 1$ per ogni I_v
- A è CONTRADDITTORIA se $I_v(A) = 0$ per ogni I_v

EQUIVALENZA LOGICA
$A \equiv B$ sse $I_v(A) = I_v(B)$ per ogni I_v

MODELLI

Definiamo ora i *modelli*, ovvero delle interpretazioni delle proposizioni atomiche, cioè una scelta di valori di verità per tutte le proposizioni atomiche.

Un modello è un sottoinsieme delle proposizioni atomiche $M \subseteq PA$ a cui è associata un'interpretazione $I_M : F_p \to \{0,1\}$ definita come tale: $I_M(p_i) = 1$ sse $p_i \in M$.
In sostanza il modello *sceglie* quali proposizioni atomiche sono vere.

Possiamo definire la relazione $M \vDash A$ tra modelli e formule che possiamo leggere come
M MODELLA A oppure come
A È SODDISFATTA IN M.
Quindi scriveremo $M \vDash A$ se A risulta vera per la scelta particolare di M.

LOGICA PROPOSIZIONALE - APPARATO DEDUTTIVO

Possiamo ora rappresentare l'apparato deduttivo della logica proposizionale.

Esso è composto da **regole di inferenza** scritte in questo modo: $\frac{A_1, ..., A_N}{B}$
Dove $A_i \in F_p$ sono le ***premesse***, mentre $B \in F_p$ è la ***conclusione***.

Alcune regole di inferenza sono ad esempio

- $\frac{A_1 \quad A_2}{A_1 \wedge A_2}$ Se ho dimostrato vere A_1 e A_2, sarà vera anche $A_1 \wedge A_2$
- $\frac{A_1 \wedge A_2}{A_1}$ Se ho dimostrato vera $A_1 \wedge A_2$, A_1 è vera
- $\frac{A \quad A \rightarrow B}{B}$ Modus Ponens
- $\frac{A \rightarrow B \quad \neg B}{\neg A}$ Modus Tollens

Le regole di inferenza sono i mattoni con i quali si costruiscono le ***dimostrazioni***.
DIMOSTRAZIONE: Catena di applicazione di regole $A_1, \ \ldots, \ A_N \vdash B$
($\vdash$ significa che da $A_1, \ \ldots, \ A_N$ si deriva B).

Abbiamo ora due nozioni distinte:

- La validità in un modello $\models$
- La derivabilità $\vdash$ introdotta perché in generale non siamo in grado di decidere direttamente la nozione semantica di validità e quindi cerchiamo di dimostrare una cosa.

Naturalmente ha senso fare le derivazioni se ciò che è derivabile è anche vero, per questo esistono questi due teoremi:

Teorema: Se $A_1, \ \ldots, \ A_n \vdash B$ allora $A_1, \ \ldots, \ A_n \models B$ *(validità, correttezza)*
Cioè se riusciamo a derivare B da $A_1, \ \ldots, \ A_n$, in ogni modello in cui sono vere $A_1, \ \ldots, \ A_n$, è vera anche B.
Questo teorema ci dice che *abbiamo scelto bene* le regole di inferenza e che queste non ci permettono di derivare cose non vere. Se questo teorema è valido, la logica è **corretta**.

Teorema: Se $A_1, \ \ldots, \ A_n \models B$ allora $A_1, \ \ldots, \ A_n \vdash B$ *(completezza)*
Cioè se in ogni modello in cui sono vere $A_1, \ \ldots, \ A_n$ ed è vera anche B, si può derivare B da $A_1, \ \ldots, \ A_n$.
Questo teorema ci dice che possiamo dimostrare tutto ciò che è vero.
Se questo teorema è valido, la logica è **completa**.

LOGICA DI HOARE

Questa logica si può vedere come costruita su due livelli diversi poiché permette di stabilire e definire il criterio di correttezza per un dato programma. In particolare, definisce questa correttezza tramite le definizioni di *precondizioni* e *postcondizioni* che sono delle formule della logica proposizionale.

PRIMO LIVELLO

Dunque al primo livello avremmo le proposizioni atomiche definite come relazioni fra le variabili del programma *($x \leq 5$, $x > y$, $z = 2^y$, ...)*.

Per dare un significato, una semantica, alle varie proposizioni atomiche definiamo la nozione di **stato della memoria**.
Supponiamo di aver scritto un programma e chiamiamo V l'insieme delle variabili del programma; uno stato della memoria è una *fotografia* della memoria del programma in un certo istante.

Possiamo definire più formalmente quest'idea di stato della memoria come una funzione $\sigma : V \to \mathbb{Z}$ che assegna un valore ad ogni variabile *(consideriamo solo variabili intere per semplicità)*.

Possiamo ora attribuire ad ogni proposizione atomica un valore di verità in un certo stato.
Ad esempio $x \leq 5$ è vera in σ se $\sigma(x) \leq 5$.

Diremo quindi che la formula α è vera in σ scrivendo $\sigma \vDash \alpha$.

SECONDO LIVELLO

Possiamo a questo punto definire le formule della logica di Hoare.
In particolare, le formule di questa logica sono costruite tramite **triple di Hoare** definite come segue

$\{\alpha\}$ C $\{\beta\}$ con α precondizione, β postcondizione e C comando/programma.

Le triple di Hoare si leggono come segue:
SE il comando C viene eseguito a partire da uno stato della memoria nel quale α è vera, **ALLORA** l'esecuzione termina e nello stato finale β è vera.

Quello che faremo sarà cercare di dimostrare varie triple di Hoare sviluppando poco per volta un apparato deduttivo per questa logica.

PSEUDO LINGUAGGIO USATO

Il linguaggio che useremo con le triple di Hoare ha le seguenti istruzioni:
x **:=** 5 *assegnamento*
if x < 0 **then**x := -x **else** x := 2*x **endif** *scelta*
while x < n **do** x := x + 1 **endwhile** *iterazione*
skip *istruzione NOP*

LOGICA DI HOARE - APPARATO DEDUTTIVO

1. Istruzione nulla $\dfrac{}{\{p\} \quad skip \quad \{p\}}$

   Iniziamo dal caso più banale, ovvero quello dell'istruzione *skip*.
   Per qualunque precondizione p, se eseguiamo uno skip questa non cambia il valore di verità della formula p poiché non cambia lo stato della memoria. Notiamo che questa regola non ha bisogno di premesse.

2. Sequenza, composizione $\dfrac{\{p\} \quad C \quad \{r\} \qquad \{r\} \quad D \quad \{q\}}{\{p\} \quad C; D \quad \{q\}}$

   Se troviamo

   - Una formula r che, sotto l'ipotesi di eseguire C a partire da uno stato in cui è valida p, risulti valida al termine dell'esecuzione
   - E che, eseguendo D da uno stato in cui è valida r alla fine risulti valida q

   Possiamo derivare la tripla che dice che se eseguiamo C e D in sequenza da uno stato in cui è valida p, alla fine varrà q.

3. Scelta $\dfrac{\{p \wedge B\} \quad C \quad \{q\} \qquad \{p \wedge \neg B\} \quad D \quad \{q\}}{\{p\} \quad if\ B\ then\ C\ else\ D\ endif \quad \{q\}}$

   Se dimostriamo che

   - Eseguendo C da uno stato dove vale la precondizione p e vale anche la condizione B al termine dell'esecuzione vale q
   - Eseguendo D da uno stato dove vale p ma non vale B arriviamo comunque ad uno stato dove vale q

   Possiamo derivare la regola della scelta dove prima dell'esecuzione dell'`if` vale p mentre dopo vale q.

4. Implicazione, conseguenza $\dfrac{\{p \to p'\} \qquad \{p'\} \quad C \quad \{q\}}{\{p\} \quad C \quad \{q\}} \qquad \dfrac{\{p\} \quad C \quad \{q\} \qquad q \to q'}{\{p\} \quad C \quad \{q'\}}$

   Osserviamo un esempio:
   Supponiamo di aver scritto un programma e di voler dimostrare $\{x > 0\} \quad C \quad \{x = 2^y\}$, cioè vogliamo dimostrare che, eseguendo il programma C a partire da uno stato della memoria dove vale $x > 0$, si arriverà in uno stato dove x sarà pari a 2^y.

   Supponiamo di aver derivato una tripla che ci dice che eseguendo C da uno stato dove vale $x \geq 0$, arriveremo in uno stato in cui $x = 2^y$, supponiamo cioè $\vdash \{x \geq 0\} \quad C \quad \{x = 2^y\}$.

   Osserviamo a questo punto che $x > 0 \to x \geq 0$; siccome quando x è maggiore di 0 è sicuramente anche maggiore o uguale a 0 è ragionevole concludere che eseguendo C da uno stato dove $x > 0$ arriverò in uno stato dove $x = 2^y$, ovvero $\vDash \{x > 0\} \quad C \quad \{x = 2^y\}$.

   Notare l'utilizzo del $\vDash$; il ragionamento fatto si riferisce alla semantica della formula.

   Generalizzando possiamo dire che se abbiamo dimostrato una tripla e osserviamo una condizione che implica la precondizione della tripla, possiamo derivare la tripla con la condizione osservata al posto della precondizione.
   Discorso analogo è valido anche per la postcondizione.

5. Assegnamento $\dfrac{}{\{q[E/x]\} \quad x{:=}E \quad \{q\}}$

   Dove con $[E/x]$ intendiamo dire *'sostituisci ogni occorenza di x in q con E'*.

   Osserviamo anche qui un esempio: prendiamo un'istruzione di assegnamento e supponiamo che il nostro obiettivo sia raggiungere uno stato dove vale $x \geq 0$, ovvero
   $x := y + 2 \quad \{x \geq 0\}$.

   Quale precondizione garantisce la postcondizione richiesta?
   Notiamo che per far sì che $x \geq 0$, y deve essere maggiore o uguale a -2.

   Possiamo generalizzare questo procedimento impostando come precondizione di un assegnamento la postcondizione desiderata sostituendo alla variabile assegnata il suo valore, cioè $\vDash \{y + 2 \geq 0\} \quad x := y + 2 \quad \{x \geq 0\}$, dove in questo caso $E = y + 2$.

   ISTANTI DI TEMPO DIVERSI
   Nulla vieta che x appaia di nuovo nella precondizione, ad esempio:
   Da $x := x + 2 \quad \{x < 0\}$ possiamo derivare $\vdash \{x + 2 < 0\} \quad x := x + 2 \quad \{x < 0\}$.
   In questo caso la x della precondizione e la x della postcondizione si riferiscono alla variabile x in due instanti di tempo diversi.

   CONTRADDIZIONI
   Supponiamo di avere $x := -2 \quad \{x > 0\}$; notiamo subito che se assegnamo -2 ad x, non potrà mai essere maggiore di 0. Applichiamo meccanicamente la regola:
   $\{-2 > 0\} \quad x := -2 \quad \{x > 0\}$ e notiamo subito che la proposizione atomica $-2 > 0$ equivale a $False$.
   Questa tripla è legittima e valida, ma non esiste alcun stato in cui valga la formula $False$.

   Anche questa regola non ha bisogno di alcuna premessa.

6. Iterazione *(correttezza parziale)* $\dfrac{\{inv \wedge B\} \quad C \quad \{inv\}}{\{inv\} \quad while\ B\ do\ C\ endwhile \quad \{inv \wedge \neg B\}}$

   L'idea per le istruzioni iterative è di trovare una formula che sia ***invariante*** rispetto al blocco dell'istruzione iterativa. Se troviamo un'invariante possiamo dire che se questa formula è vera all'inizio dell'istruzione iterativa, lo sarà anche alla fine e in più possiamo dire che al termine del blocco iterativo vale anche la negazione della condizione di ciclo.
   In generale, avendo un'istruzione $W =$ `while B do C end` $-$ `while`, se deriviamo $\vdash \{p \wedge B\}\ C\ \{p\}$, allora possiamo dire che p è invariante rispetto a W.

   ALCUNE NOTE

   - Data un'istruzione iterativa, non c'è un solo invariante. Ad esempio, $True$ è invariante per ogni istruzione iterativa. Generalmente ci interessano gli invarianti utili alla dimostrazione in corso.
   - Nella pratica, le istruzioni iterative sono inserite in programmi, di conseguenza la scelta dell'invariante dipende dal contesto e si abbina alla regola dell'implicazione.

     Vediamo un esempio:
     Supponiamo di avere la seguente formula $\{p\}\ C\ \{r\}\ W\ \{z\}\ D\ \{q\}$ e di volerci ricondurre a $\{inv\}\ W\ \{inv \wedge \neg B\}$; se riusciamo a dimostrare che $r \rightarrow inv$ e che $inv \neg B \rightarrow z$ avremo ricostruito una dimostrazione completa.
   - Avvolte l'invariante non è assoluto, ma lo diventa aggiungendoci la condizione di ciclo.
   - L'invariante in genere è violata guardando lo stato della memoria durante l'esecuzione del blocco, ma viene ripristinata al termine del blocco.

7. Iterazione *(correttezza totale)* $\dfrac{inv \to E \geq 0 \qquad \{inv \wedge B \wedge E = k\} \quad C \quad \{inv \wedge E < k\}}{\{inv\} \quad while\ B\ do\ C\ endwhile \quad \{inv \wedge \neg B\}}$

   Quando trattiamo istruzioni iterative esistono due modi diversi per cui l'istruzione può fallire rispetto alla specifica data da una tripla:

   1. L'istruzione iterativa, eseguita a partire da uno stato che soddisfa la precondizione, termina in uno stato che *NON* soddisfa la postcondizione *(gestito nella correttezza parziale)*
   2. L'istruzione iterativa *NON* termina

   Chiaramente per dimostrare la correttezza di un programma è necessario tener traccia di entrambi questi casi. Parleremo quindi di *correttezza totale* quando considereremo anche la terminazione e di *correttezza parziale* quando ci concentreremo solo su pre e postcondizioni dando per scontato che l'istruzione termini.

   Consideriamo $W = \texttt{while}\ B\ \texttt{do}\ C\ \texttt{endwhile}$

   TECNICA DI DIMOSTRAZIONE
   Supponiamo che E sia un'espressione aritmetica nella quale compaiono variabili del programma, costanti numeriche e operazioni aritmetiche e che inv sia un invariante di ciclo per W scelta in modo che:

   1. $inv \to E \geq 0$
   2. $\vdash^{TOT}_{ITER} \{inv \wedge B \wedge E = k > 0\} \quad C \quad \{inv \wedge E < k\}$

   ALLORA:
   $\vdash^{TOT}_{ITER} \{inv\} \quad W \quad \{inv \wedge \neg B\}$

   Notiamo il ^{TOT}; questa una definizione ricorsiva perché C potrebbe contenere al suo interno altre operazioni ricorsive.

   Cerchiamo cioè una quantità *variante* che sarà maggiore o uguale a 0 quando è valido l'invariante e che permetterà di derivare la tripla che dice che il variante E avrà un valore costante $k > 0$ prima dell'esecuzione e sarà $E < k$ dopo l'esecuzione. Questo significa che se eseguiamo il corpo dell'iterazione C partendo da uno stato dove valgono inv, B e E ha un certo valore, al termine di una iterazione il valore di E diminuisce strettamente.

   Notiamo che l'invariante implica che $E \geq 0$; avendo che ad ogni iterazione il valore di E diminuisce strettamente e che l'invariante è sempre valido prima e dopo l'iterazione, siamo sicuri che l'istruzione prima o poi terminerà.

   OSSERVAZIONI

   - E ***non*** è una formula logica, $E \geq 0$ lo è
   - Lo 0 in $E \geq 0$ può essere sostituito da qualsiasi numero

PROPRIETÀ GENERALI DELLA LOGICA DI HOARE

La logica di Hoare è una logica ***corretta*** $\vdash \implies \vDash$; questo significa che tutte le triple che riusciamo a derivare sono sicuramente delle triple valide.
È anche ***completa*** *(relativamente)* $\vDash \implies \vdash$; tutto ciò che è valido è derivabile. Tuttavia è una completezza relativa a causa dell'incompletezza dell'aritmetica: potrebbe succedere di dover dimostrare una proprietà aritmetica che però è indimostrabile *(caso molto remoto)*.

La logica di Hoare può aiutarci anche a ragionare sui programmi: supponiamo di avere un comando C e una formula q e di voler trovare una formula p tale che $\{p\} \quad C \quad \{q\}$

Esempio: $x := k;\ y := 2x; \quad \{y > 0\}$
Soluzioni: $k = 5 \qquad k = 12 \qquad k > 3 \qquad k > 0$

Esiste una precondizione *'migliore'*? Come possiamo definirla? Come possiamo calcolarla?
Notiamo che tra le soluzioni proposte $k > 0$ è quella che dà più libertà di scelta di uno stato iniziale per l'esecuzione del comando.
Notiamo anche che se tale soluzione non fosse vera, cioè se $k \leq 0$, allora y non potrebbe mai essere > 0; possiamo quindi dire che quella sia la formula più inclusiva e cioè quella formula che se viene violata non permetterà di raggiungere la postcondizione.

ALCUNE NOZIONI

Fissiamo un programma C e chiamiamo

- V l'insieme delle variabili di C
- $\Sigma = \{\sigma \mid \sigma : V \to \mathbb{Z}\}$ l'insieme degli stati della memoria
- Π l'insieme delle formule p su V *(come $x = 3$, $> y$, ecc)*
- $\vDash \ \subseteq \ \Sigma \times \Pi$ la relazione per dire se una formula è vera in un certo stato
  $\sigma \vDash p$ *"p è vera in σ"*
- $t(\sigma) = \{p \in \Pi \mid \sigma \vDash p\}$ una funzione che associa ad uno stato σ tutte le formule che sono vere in esso $\sigma \vDash p$. Questa funzione non può restituire un insieme vuoto; per ogni stato c'è almeno una formula che lo soddisfa *(ad esempio una formula che impone il valore di ogni variabile del programma)*.
- $m(p) = \{\sigma \in \Sigma \mid \sigma \vDash p\}$ una funzione che associa ad una formula tutti gli stati che la soddisfano. Questa funzione può restituire un insieme vuoto; nessuno stato soddisfa una formula contraddittoria.

Possiamo estendere le ultime due funzioni dichiarate ai sottoinsiemi facendo uso di $S \subseteq \Sigma$ sottoinsieme di stati e $F \subseteq \Pi$ sottoinsieme di formule definendo:

- $t(S) = \{p \in \Pi \mid \forall s \in S : s \vDash p\} = \bigcap_{s \in S} t(s)$ è le formule comuni a *tutti gli stati* di S, ovvero l'intersezione delle immagini calcolate sui singoli stati s.
- $m(F) = \{s \in \Sigma \mid \forall p \in F : s \vDash p\} = \bigcap_{p \in F}$ è l'insieme degli stati tali per cui *tutte le formule* in F sono valide, ovvero l'intersezione delle immagini calcolate sulle singole formule f.

OSSERVAZIONI

- $S \subseteq m(t(s)) \qquad$ e $\qquad F \subseteq t(m(s))$
- Se $A \subseteq B$, allora $m(B) \subseteq m(A)$

LOGICA PROPOSIZIONALE E INSIEMI DI STATI

Sappiamo che c'è una relazione forte tra la logica proposizionale e la teoria degli insiemi, in particolare:

- $m(\neg p) = \Sigma \setminus m(p)$
- $m(p \vee q) = m(p) \cup m(q)$
- $m(p \wedge q) = m(p) \cap m(q)$
- $m(p \rightarrow q) = \Sigma - m(p) \cup m(q)$

IMPLICAZIONE
Notiamo che possiamo vedere l'implicazione in due ottiche diverse:

- **CONNETTIVO LOGICO**: in questo caso non è che una abbreviazione di una formula più lunga, ovvero $p \rightarrow q \equiv \neg p \vee q$, e quindi $m(p \rightarrow q) = m(\neg p) \cup m(q)$.
- **RELAZIONE TRA FORMULE**: in questo caso abbiamo che se p implica q, allora $m(p) \subseteq m(q)$, che equivale a dire che q è ***più debole*** di p, poiché q coincide ad un maggior numero di stati e quindi ci dà meno informazioni su quale sia lo stato corrente.

CRITERIO DI SCELTA DELLA PRECONDIZIONE MIGLIORE

Il criterio di scelta che andremo ad utilizzare è la **weakest precondition**, ovvero, fissati p e q, cercheremo la precondizione più debole, che coincide al maggior numero di stati, tale che $\vDash \{p\} \quad C \quad \{q\}$.

Questa precondizione più debole corrisponde al più grande insieme di stati dai quali l'esecuzione di C porta ad uno stato in $m(q)$.

È stato dimostrato che esiste sempre la precondizione più debole per un comando e una postcondizione. Essa può essere l'insieme di tutti gli stati o anche l'insieme vuoto.

NOTAZIONE: $wp(C,\ q)$ precondizione più debole per $C \quad \{q\}$

Teorema: $\vDash \{p\} \quad C \quad \{q\}$ se e solo se $p \rightarrow wp(C, q)$

REGOLE DI CALCOLO DI WP

Vediamo come calcolare le precondizioni più deboli. Lo vedremo separatamente per ogni tipo di istruzione.

- **Assegnamento**: $wp(x := E,\ q) = q[E/x]$ Nel caso dell'assegnamento la precondizione più debole coincide con la sostituzione effettuata dalla regola di derivazione.
- **Sequenza**: $wp(C_1,\ C_2,\ q) = wp(C_1,\ wp(C_2,\ q))$ Per calcolare la precondizione più debole per una sequenza, calcoliamo prima la wp del comando più interno e procediamo a ritroso.
- **Scelta**: C : `if` B `then` C_1 `else` C_2 `endif`
  $wp(C,\ q) = (B \wedge wp(C_1,\ q)) \vee (\neg B \wedge (C_2,\ q))$
  La precondizione più debole in questo caso sarà l'*unione* delle precondizioni più deboli dei due rami dell'`if`, combinati con la relativa formula di scelta.

  Esempio concreto:
  Consideriamo il programma P:

  ```
  if y == 0 then
      x := 0;          //C
  else
      x := x * y;      //D
  end-if
  ```

  Vogliamo calcolare $wp(P,\ x = y)$.
  Per farlo dobbiamo calcolare $wp(C,\ x = y)$ e $wp(D,\ x = y)$.

  Notiamo che si tratta di due assegnamenti, quindi:
  $wp(C,\ x = y) \implies 0 = y$
  $wp(D,\ x = y) \implies xy = y \implies y = 0 \vee x = 1$

  Possiamo ora calcolare $wp(P,\ x = y) \implies (y = 0 \wedge y = 0) \vee (y \neq 0 \wedge y = 0 \vee x = 1)$
  $(y = 0 \wedge y = 0) \vee (y \neq 0 \wedge y = 0 \vee x = 1) \implies (y = 0) \vee (y \neq 0 \wedge x = 1)$
- **Iterazione**: $wp(W,\ q) = (\neg B \wedge q) \vee (B \wedge wp(C,\ W,\ q)) \equiv$
  $(\neg B \wedge q) \vee (B \wedge wp(C,\ wp(W,\ q)))$

  Esempio concreto:
  Consideriamo il programma P:

  ```
  while x > 0 do          //W
      x := x - 1      //C //W
  end-while               //W
  ```

  Vogliamo calcolare $wp(W,\ x = 0)$
  Ragioniamo così: l'istruzione iterativa può:

  - Non essere mai eseguita e quindi nello stato iniziale B non vale. In questo caso la precondizione più debole equivale a $\neg B \wedge q$, poiché è come se il ciclo fosse uno skip che non cambia la memoria; è necessario che q sia valido all'inizio.
  - Nello stato iniziale è vera B: il programma equivale all'esecuzione di C una volta e alla replicazione di W: possiamo pensare di dare una definizione ricorsiva di wp. Non c'è quindi un algoritmo *meccanico* per trovare una precondizione più debole per un ciclo.

CONTESTI DIVERSI

Abbiamo visto che una tripla di Hoare possa garantire la correttezza di un programma ma anche come possa portarci a concetti come la precondizione più debole. Vediamo ora gli stessi concetti delle triple di Hoare in contesti diversi.

DESIGN BY CONTRACT

Il primo contesto che andiamo ad osservare appartiene all'ingegneria del software; le triple possono essere interpretate nell'ambito di una strategia di sviluppo di programmi chiamata *design by contract* o *programming by contract*.

L'idea sottostante è che lo sviluppo di un programma corrisponde ad un contratto fra due contraenti *cliente* e *fornitore*; diciamo che il *fornitore* fornisce il programma e il *cliente* specifica cosa vorrebbe che facesse il programma.
In questa visione una tripla è un contratto tra *fornitore* e *cliente*, ovvero un accordo nel quale ciascuna parte si assume delle obbligazioni e in cambio ha dei diritti.

Interpretiamo le triple come: $\{p\} \quad C \quad \{q\}$, dove p rappresenta l'obbligazione del cliente, C la funzione *invocata* dal cliente e q l'obbligazione del fornitore.

Alcuni linguaggi incorporano nella propria sintassi l'idea di definire i termini del contratto, ad esempio ***Eiffel***.
Esistono anche strumenti software, come ***Java Modelling Language*** che permettono di incorporare questa idea anche in linguaggi come Java usando delle speciali annotazioni.

STORIA

Le prime idee sulla correttezza dei programmi nacquero nei primi anni 70 dai lavori di **R. W. Floyd**, **C. A. R. Hoare** e **E. W. Dijkstra** *(che ha introdotto il concetto di precondizione più debole)*.

Le idee di **Dijkstra** sono state usate anche per definire una semantica dei programmi, e dare quindi un *significato* ad un programma. Si possono dare diverse interpretazioni, una possibile è: il significato di un programma è una funzione che va dallo spazio dei possibili dati in ingresso allo spazio dei possibili valori in uscita. Si può anche dare un significato operazionale dove il significato di un programma è una sequenza di cambi dello stato della memoria provocati dall'esecuzione del programma.

DIMOSTRAZIONI DI CORRETTEZZA

Abbiamo visto dimostrazioni applicabili ad un linguaggio molto semplice che, seppur dal punto di vista teorico è completo, dal punto di vista pratico è molto limitato. Pensiamo alle estensioni pratiche di un linguaggio come *funzioni*, *ricorsione*, *tipi strutturati*, *classi e oggetti* e a come queste possano influire sulle dimostrazioni di correttezza.

Un caso dove la dimostrazione di correttezza si fa decisamente più complicata è il caso dei programmi concorrenti; la logica di Hoare si basa sul fatto che i cambiamenti dello stato della memoria sono deterministici, ma questo non è più vero nel caso di programmi concorrenti. Inoltre, non sempre i programmi concorrenti hanno uno stato finale *(basti pensare ad un sistema operativo)* e si ritrovano ad interagire continuamente con l'ambiente: questi sono tutti aspetti da considerare per dimostrare la correttezza di un programma concorrente.

LOGICA DI HOARE E SINTESI DI PROGRAMMI ITERATIVI

Possiamo usare la logica di Hoare anche per sviluppare dei programmi invece che validarli.
Supponiamo di voler calcolare un'approssimazione intera della radice quadrata di un numero k.

1. La tripla obiettivo sarà dunque $\{k \geq 0\} \quad P \quad \{0 \leq x^2 \leq k < (x+1)^2\}$
2. Ipotizziamo la struttura generale del programma: calcoliamo il valore per approssimazioni successive iterando finché il valore non soddisfa la postcondizione.

```
x := E(k);          //Il valore iniziale dipende da k
while B(x, k) do    //La condizione di ciclo dipende da x e k
    x := F(x, k)    //Il nuovo valore dipene dal vecchio valore e da k
end-while
```

3. Spezziamo la postcondizione per cercare un invariante di ciclo
   $0 \leq x^2 \qquad x^2 \leq k \qquad k < (x+1)^2$

   Notiamo che nel corso dell'iterazione vorremo che x^2 rimanga sempre $\leq k$, mentre $(x+1)^2 > k$ non può essere un invariante poiché è la caratteristica della soluzione cercata.

   Dunque $0 \leq x^2 \land x^2 \leq k$ è un possibile invariante.
   $x := E(k)$ deve stabilire l'invariante; poiché $k \geq 0$ *(precondizione generale)*, possiamo scegliere 0 come valore iniziale di x.
4. Se $inv = 0 \leq x^2 \leq k$, al termine dell'esecuzione varrà $inv \land \neg B(x,k)$, quindi dobbiamo scegliere $B(x,k)$ in modo che $(inv \land \neg B(x,k)) \rightarrow (inv \land k < (x+1)^2)$ *(ovvero che implichi la postcondizione generale)*.

   Poniamo dunque $B(x,k) = (x+1)^2 \leq k$.
5. Ora osserviamo che se nel corpo dell'iterazione incrementiamo x, prima o poi $(x+1)^2$ sarà maggiore di k. Il programma potrebbe quindi essere:

```
x := 0;
while (x+1)*(x+1) <= k do
    x := x + 1;
end-while
```

SCHEMA GENERALE DI DIMOSTRAZIONE

Supponiamo di dover dimostrare $\{p\} \quad V;W;Z \quad \{q\}$, supponendo che V e W non contengano istruzioni iterative.
Uno schema generale è:

1. Ricaviamo da $Z \quad \{q\}$ la $wp(Z,q) \equiv s$ $\qquad\qquad$ $\{s\} \quad Z \quad \{q\}$
2. Cerchiamo un'invariante i per W tale che $(i \land \neg B) \rightarrow s$ $\qquad\qquad$ $\{i\} \quad W;Z \quad \{q\}$
3. Cerchiamo una formula u tale che $\vdash \{p\} \quad V \quad \{u\}$ e $u \rightarrow i$ $\qquad\qquad$ $\{p\} \quad V;W;Z \quad \{q\}$

ESEMPI DI DERIVAZIONE

ASSEGNAMENTO

Consideriamo l'istruzione $x := -x$, fissando come postcondizione $\{x < 0\}$.
Ricaviamo una precondizione corretta per questa combinazione di comando e postcondizione.

Applichiamo la regola dell'assegnamento: $\vdash_{ASS} \{-x < 0\} \quad x := -x \quad \{x < 0\}$
Possiamo riscrivere la tripla in maniera più leggibile come $\{x > 0\} \quad x := -x \quad \{x < 0\}$.

ASSEGNAMENTO

Consideriamo $x := 2 * y \quad \{x > y\}$
$\vdash_{ASS} \{2 * y > y\} \quad x := 2 * y \quad \{x > y\} \quad \equiv \quad \{y > 0\} \quad x := 2 * y \quad \{x > y\}$

CONTROESEMPI

Supponiamo ora di dover dimostrare questa tripla ? $\quad \{True\} \quad x = 2 * y \quad \{x > y\}$
Proviamo ad applicare la regola dell'assegnamento $\vdash_{ASS} \{y > 0\} \quad x = 2 * y \quad \{x > y\}$:
sappiamo ora che questa tripla è valida poiché l'abbiamo ottenuta tramite la derivazione, ma il nostro obiettivo è la prima tripla. L'unica regola che possiamo pensare di applicare è quella dell'implicazione, ma osserviamo che non è vero che $True \rightarrow y > 0$, dunque non possiamo dimostrare la validità della prima tripla.
Questo è un bene in realtà, poiché non è certamente vero che eseguendo quell'assegnamento da uno stato in cui vale $True$ arriverò *sicuramente* in uno stato in cui vale $x > y$.

Tuttavia, il fatto che non siamo riusciti a dimostrare che sia vera non vuol dire che sia falsa: dobbiamo dimostrare che è falsa. Il modo più semplice è trovare un controesempio.
Un controesempio in questo caso è uno stato in cui $y = -1$: applicando l'assegnamento avremo $x = -2$ e $x \not> y$.

SEQUENZA

Supponiamo di voler dimostrare ? $\quad \{x + y = K\} \quad x := x - 1;\ y := y + 1 \quad \{x + y = K\}$
Notiamo subito che qui il programma è composto da una sequenza di due assegnamenti, quindi dobbiamo trovare una formula intermedia che faccia da postcondizione per il primo assegnamento e da precondizione per il secondo. Siccome si tratta di assegnamenti, possiamo pensare di usare la regola dell'assegnamento, ottenendo:

- Da $y := y + 1 \quad \{x + y = K\} \qquad \vdash_{ASS} \{x + y + 1 = K\} \quad y := y + 1 \quad \{x + y = K\}$
- E a questo punto da $x := x - 1 \quad \{x + y + 1 = K\}$
  $\vdash_{ASS} \{x - 1 + y + 1 = K\} \quad x := x - 1 \quad \{x + y + 1 = K\}$

Notiamo che $x - 1 + y + 1 = K \equiv x + y = K$.

Notiamo anche che la precondizione dell'assegnamento $y := y + 1$ è uguale alla postcondizione dell'assegnamento $x := x - 1$, dunque possiamo applicare la regola della sequenza
$\vdash_{SEQ} \{x + y = K\} \quad x := x - 1;\ y := y + 1 \quad \{x + y = K\}$

Abbiamo così dimostrato la validità della tripla iniziale.

SCELTA

Consideriamo il programma P

```
y := k;
if x > 0 then
    y := y * x
else
    y := -2 * x
end-if
```

Vogliamo stabilire se vale ? $\{k > 0\} \quad P \quad \{y \geq 0\}$.

Osserviamo la postcondizione e in particolare che questa vale subito dopo un'istruzione di scelta: per poter derivare una tripla come istruzione di scelta dobbiamo *"spezzare"* i rami della scelta e stabilire se tutti e due i casi portino alla postcondizione richiesta.
I rami in questo caso sono

- $y := y * x \quad \{y \geq 0\}$
  Possiamo applicare questo ramo la regola dell'assegnamento
  $\vdash_{ASS} \{y * x \geq 0\} \quad y := y * x \quad \{y \geq 0\}$
- $y := -2 * x \quad \{y \geq 0\}$
  Applichiamo ancora la regola dell'assegnamento
  $\vdash_{ASS} \{-2 * x \geq 0\} \quad y := -2 * x \quad \{y \geq 0\}$

Ricordiamo che le precondizioni delle premesse della regola della scelta sono
$\{p \wedge B\}$ e $\{p \wedge \neg B\}$.

Notiamo che la precondizione della tripla da dimostrare dice che $k > 0$ e l'istruzione prima dell' **`if`** assegna k a y, di conseguenza avremo che sicuramente $y > 0$ dopo il primo assegnamento.

Scriviamo ora, sostituendo a $p = y > 0$ e a $B = x > 0$ *(la condizione dell'`if`)* le premesse della regola della scelta *(dobbiamo dimostrare queste triple per poterla applicare)*

- $? \quad \{y > 0 \wedge x > 0\} \quad y := y * x \quad \{y \geq 0\}$ $\qquad$ *(*$\{p \wedge B\}\ C\ \{q\}$*)*
- $? \quad \{y > 0 \wedge x \leq 0\} \quad y := -2 * x \quad \{y \geq 0\}$ $\qquad$ *(*$\{p \wedge \neg B\}\ D\ \{q\}$*)*

Notiamo che la nostra precondizione nel primo ramo ($y * x \geq 0$) non coincide esattamente con la precondizione ricavata ora, ma notiamo anche che $y > 0 \wedge x > 0 \rightarrow xy > 0$, quindi
$\vdash_{IMPL} \{y > 0 \wedge x > 0\} \quad y := y * x \quad \{y \geq 0\}$

Osserviamo ora il secondo ramo; dobbiamo confrontare $y > 0 \wedge x \leq 0$ con $-2x \geq 0$.
Notiamo che possiamo riscrivere $-2x \geq 0$ come $x \leq 0$.
Senz'altro $y > 0 \wedge x \leq 0 \rightarrow x \leq 0$, quindi applichiamo nuovamente la regola dell'implicazione
$\vdash_{IMPL} \{y > 0 \wedge x \leq 0\} \quad y := -2x \quad \{y \geq 0\}$

Notiamo ora che le ultime due triple derivate hanno esattamente la forma delle premesse della regola della scelta, quindi
$\vdash_{IF} \{y > 0\}\ if\ x > 0\ then\ y := y * x\ else\ y := -2 * x\ endif\ \{y \geq 0\}$

Ora manca solo da dimostrare che anche l'assegnamento iniziale è valido: per fare ciò usiamo la regola dell'assegnamento usando come postcondizione la precondizione dell'**`if`**, quindi
$\vdash_{ASS} \{k > 0\} \quad y := k \quad \{y > 0\}$

Infine, osservando che la postcondizione dell'assegnamento è la precondizione dell'**`if`** possiamo dire $\vdash_{SEQ} \{k > 0\} \quad P \quad \{y \geq 0\}$.

ITERAZIONE - ESPONENZIALE

Consideriamo il seguente programma P che calcola l'esponenziale

```
i := 0; s := 1; //A
while i < N do      //W
    i := i + 1; //C //W
    s := s * x; //C //W
end-while           //W
```

Vogliamo dimostrare la tripla ? $\{N \geq 0\} \quad P \quad \{s = x^N\}$

Notiamo che il programma contiene due assegnamenti e un ciclo, dobbiamo quindi cercare un'invariante; per farlo proviamo a simulare i primi passi di un'esecuzione:
$i \quad 0 \quad 1 \quad 2 \quad 3 \quad \dots$
$s \quad 1 \quad x \quad x^2 \quad x^3 \quad \dots$

Notiamo che $s = x^i$ per ogni iterazione; ogni volta i viene incrementato, ma anche s viene aggiornato. Chiaramente, se dovessimo osservare la memoria dopo il primo assegnamento ma prima dell'aggiornamento di s l'invariante non sarebbe più valido, ma a noi interessano i momenti iniziale e finale del blocco iterativo.

1. DIMOSTRAZIONE INVARIANTE
Ipotizziamo quindi che $s = x^i$ sia un'invariante: dobbiamo dimostrarlo e per farlo possiamo usare la tripla che fa da premessa alla regola dell'iterazione *(che è sostanzialmente la definizione di invariante)*, ovvero ? $\{s = x^i \wedge i < N\} \quad C \quad \{s = x^i\}$

Notiamo che la C è composto da due assegnamenti, quindi possiamo derivare la tripla premessa della regola dell'iterazione usando la regola dell'assegnamento tramite $C \quad \{s = x^i\}$:
$\vdash_{ASS} \{sx = x^i\} \quad s := s * x \quad \{s = x^i\}$

Usiamo ora la precondizione ottenuta come postcondizione per l'altro assegnamento:
$\vdash_{ASS} \{sx = x^{i+1}\} \quad i := i + 1 \quad \{sx = x^i\}$

A questo punto, osservando che la precondizione della prima tripla derivata è la postcondizione della seconda, possiamo usare la regola della sequenza per derivare
$\vdash_{SEQ} \{sx = x^{i+1}\} \quad C \quad \{sx = x^i\}$.

Notiamo però che i due assegnamenti sono indipendenti, ovvero operano su variabili distinte; questo significa che avremmo anche potuto applicare la regola dell'assegnamento contemporaneamente ad entrambi gli assegnamenti ottenendo lo stesso risultato.

A questo punto abbiamo una tripla con la postcondizione che ci serviva ma con la precondizione che non coincide esattamente con quella della tripla da dimostrare, quindi dobbiamo manipolarla per ricondurla a quella che ci serve.
Notiamo che $sx = x^{i+1} \implies sx = x^i x \implies s = x^i$
Otteniamo quindi $\vdash \{s = x^i\} \quad C \quad \{s = x^i\}$.

A noi però serve derivare la precondizione $s = x^i \wedge i < N$, ma osserviamo che
$s = x^i \wedge i < N \rightarrow s = x^i$, quindi possiamo derivare, tramite la regola dell'implicazione
$\vdash_{IMPL} \{s = x^i \wedge i < N\} \quad C \quad \{s = x^i\}$

Abbiamo quindi dimostrato l'invariante e derivato la premessa per la regola dell'iterazione.
Possiamo ora applicare tale regola $\vdash_{ITER} \{s = x^i\} \quad W \quad \{s = x^i \wedge i \geq N\}$.

2. RAFFORZAMENTO INVARIANTE

Notiamo però che quello che vogliamo derivare noi ha una postcondizione $s = x^N$, mentre quello che abbiamo ricavato con la regola dell'iterazione è più debole: abbiamo ricavato che $s = x^i$ e che $i \geq N$.
Per avere la postcondizione voluta dobbiamo dimostrare che al termine dell'esecuzione $i = N$, ma l'invariante trovato non è sufficiente a questo scopo e quindi dobbiamo *rafforzarlo*.

Per farlo, osserviamo che anche $i \leq N$ è un invariante, quindi ipotizziamolo:
$? \quad \{i \leq N \wedge i < N\} \quad C \quad \{i \leq N\}$ Notiamo che $i \leq N \wedge i < N \equiv i < N$

Usiamo sempre la regola dell'assegnamento:
$\vdash_{ASS} \{i + 1 \leq N\} \quad C \quad \{i \leq N\}$

Notiamo che $i < N \rightarrow i + 1 \leq N$:
$\vdash_{IMPL} \{i < N\} \quad C \quad \{i \leq N\}$

Possiamo ora derivare, tramite la regola dell'iterazione:
$\vdash_{ITER} \{i \leq N\} \quad W \quad \{i \leq N \wedge i \geq N\}$ Notiamo che $i \leq N \wedge i \geq N \equiv i = N$

Ora combiniamo i due invarianti:
$\vdash_{ITER} \{s = x^i \wedge i \leq N\} \quad W \quad \{s = x^i \wedge i \leq N \wedge i \geq N\}$, ovvero
$\vdash \{s = x^i \wedge i \leq N\} \quad W \quad \{s = x^i \wedge i = N\}$, ovvero
$\vdash \{s = x^i \wedge i \leq N\} \quad W \quad \{s = x^N\}$

3. ASSEGNAMENTI INIZIALI

A questo punto ci rimane solo da dimostrare che dopo gli assegnamenti iniziali valgono gli invarianti
$? \quad \{N \geq 0\} \quad A \quad \{s = x^i \wedge i \leq N\}$

Usiamo la regola dell'assegnamento:
$\vdash_{ASS} \{1 = x^i \wedge i \leq N\} \quad s := 1 \quad \{s = x^i \wedge i \leq N\}$

$\vdash_{ASS} \{1 = x^0 \wedge 0 \leq N\} \quad i := 0 \quad \{1 = x^i \wedge i \leq N\}$, ovvero
$\vdash \{N \geq 0\} \quad i := 0 \quad \{1 = x^i \wedge i \leq N\}$

A questo punto usiamo la regola della sequenza:
$\vdash_{SEQ} \{N \geq 0\} \quad A \quad \{s = x^i \wedge i \leq N\}$

4. DIMOSTRAZIONE FINALE

A questo punto possiamo combinare, con la regola della sequenza, le triple
$\{N \geq 0\} \quad A \quad \{s = x^i \wedge i \leq N\}$ e $\{s = x^i \wedge i \leq N\} \quad W \quad \{s = x^N\}$, ottenendo
$\vdash_{SEQ} \{N \geq 0\} \quad P \quad \{s = x^N\}$, dimostrando così la tripla iniziale.

ITERAZIONE - DIVISIONE

Consideriamo il programma P che calcola quoziente e resto di x/y

```
quo := 0; rem := x;     //A
while rem >= y do       //W
    rem := rem - y; //C //W
    quo := quo + 1; //C //W
end-while               //W
```

Vogliamo dimostrare ? $\{x \geq 0 \wedge y \geq 0\} \quad P \quad \{x = quo * y + rem \wedge rem \geq 0 \wedge rem < y\}$

Osservazione: $rem < y \equiv \neg B$

Notiamo che siamo in presenza di un ciclo, cerchiamo dunque un'invariante:
Simuliamo qualche passo con $x = 26$ e $y = 5$

quo	0	1	2	3	4	5
rem	26	21	16	11	6	1

Osserviamo sia la tabella che la postcondizione di interesse.

1. DIMOSTRAZIONE INVARIANTE

Ipotesi: $x = quo * y + rem \wedge rem \geq 0$ è invariante.
$? \quad \{x = quo * y + rem \wedge rem \geq 0 \wedge rem \geq y\} \quad C \quad \{inv\}$

Usiamo la regola dell'assegnamento:
$\vdash_{ASS} \{x = (quo + 1) * y + rem - y \wedge rem - y \geq 0\} \quad C \quad \{inv\}$, ovvero
$\vdash \{x = quo * y + rem \wedge rem \geq y\} \quad C \quad \{inv\}$

Sappiamo però che $y \geq 0$, quindi
$\vdash \{x = quo * y + rem \wedge rem \geq 0\} \quad C \quad \{inv\}$

A questo punto possiamo usare la regola dell'iterazione:
$\vdash_{ITER} \{x = quo * y + rem \wedge rem \geq 0\} \quad W \quad \{x = quo * y + rem \wedge rem \geq 0 \wedge rem < y\}$

In questo caso abbiamo già ottenuto la postcondizione che ci interessa.

2. ASSEGNAMENTI INIZIALI

Per concludere la dimostrazione dobbiamo ora dimostrare che l'invariante vale dopo gli assegnamenti iniziali:
$\vdash_{ASS} \{x = x \wedge x \geq 0\} \quad A \quad \{x = quo * y + rem \wedge rem \geq 0\}$, ovvero
$\vdash \{x \geq 0\} \quad A \quad \{x = quo * y + rem \wedge rem \geq 0\}$

Notiamo che $x \geq 0 \wedge y \geq 0 \rightarrow x \geq 0$, quindi
$\vdash_{IMPL} \{x \geq 0 \wedge y \geq 0\} \quad A \quad \{x = quo * y + rem \wedge rem \geq 0\}$

Dunque, l'invariante è valido dopo gli assegnamenti iniziali.

3. DIMOSTRAZIONE FINALE

Ora possiamo comporre il tutto con la regola della sequenza e terminare la dimostrazione
$\vdash_{SEQ} \{x \geq 0 \wedge y \geq 0\} \quad P \quad \{x = quo * y + rem \wedge rem \geq 0 \wedge rem < y\}$

CORRETTEZZA TOTALE - ESEMPIO

Consideriamo il programma P

```
while x > 5 do
    x := x - 1;
end-while
```

Supponiamo di voler dimostrare ? $\{x > 5\} \quad P \quad \{x = 5\}$

Dobbiamo cercare un variante E il cui valore decresce ad ogni esecuzione dell'iterazione e un invariante inv che garantisca che E abbia sempre valore maggiore o uguale a 0.

In questo caso possiamo usare $E = x - 5$ e $inv = x \geq 5$. Osserviamo che $inv \rightarrow E \geq 0$.

Dimostriamo l'invariante:
$? \quad \{x \geq 5 \wedge x > 5\} \quad C \quad \{x \geq 5\}$

Usiamo la regola dell'assegnamento:
$\vdash_{ASS} \{x \geq 6\} \quad x := x - 1 \quad \{x \geq 5\}$

Osserviamo che $x \geq 5 \wedge x > 5 \equiv x \geq 6$:
$\vdash \{x \geq 5 \wedge x > 5\} \quad C \quad \{x \geq 5\}$ dunque l'invariante è valido.

Ora dobbiamo dimostrare ? $\{inv \wedge B \wedge x - 5 = k\} \quad x := x - 1 \quad \{inv \wedge x - 5 < k\}$

Applichiamo la regola dell'assegnamento:
$\vdash_{ASS} \{x - 1 \geq 5 \wedge x - 1 - 5 < k\} \quad x := x - 1 \quad \{x \geq 5 \wedge x - 5 < k\}$ ovvero
$\vdash \{x \geq 6 \wedge x - 6 < k\} \quad x := x - 1 \quad \{x \geq 5 \wedge x - 5 < k\}$

Notiamo che $x \geq 5 \wedge x > 5 \wedge x - 5 = k \rightarrow x \geq 6 \wedge x - 6 < k$, poiché se $x > 5$, allora sicuramente $x >= 6$ visto che lavoriamo con interi e se $x - 5 = k$ sicuramente $x - 6 < k$.
$\vdash_{IMPL} \{x \geq 5 \wedge x > 5 \wedge x - 5 = k\} \quad x := x - 1 \quad \{x \geq 5 \wedge x - 5 < k\}$

A questo punto possiamo usare la regola dell'iterazione totale
$\vdash^{TOT}_{ITER} \{x \geq 5\} \quad P \quad \{x \geq 5 \wedge x \leq 5\}$

E se B fosse $x \neq 5$?
Proviamo a vedere se l'invariante di prima è ancora valido
Dovremmo quindi dimostrare ? $\{x \geq 5 \wedge x \neq 5\} \quad C \quad \{x \geq 5\}$

Usiamo sempre la regola dell'assegnamento
$\vdash_{ASS} \{x \geq 6\} \quad x := x - 1 \quad \{x \geq 5\}$

Osserviamo che anche in questo caso $x \geq 5 \wedge x \neq 5 \equiv x \geq 6$, quindi
$\vdash \{x \geq 5 \wedge x \neq 5\} \quad C \quad \{x \geq 5\}$, quindi l'invariante è ancora valido.

Dimostriamo ora ? $\{x \geq 5 \wedge x \neq 5 \wedge x - 5 = k\} \quad C \quad \{inv \wedge x - 5 < k\}$
Applichiamo la regola dell'assegnamento:
$\vdash_{ASS} \{x \geq 6 \wedge x - 6 < k\} \quad x := x - 1 \quad \{x \geq 5 \wedge x - 5 < k\}$

Notiamo che $x \geq 5 \wedge x \neq 5 \wedge x - 5 = k \rightarrow x \geq 6 \wedge x - 6 < k$, quindi
$\vdash_{IMPL} \{x \geq 5 \wedge x \neq 5 \wedge x - 5 = k\} \quad C \quad \{x \geq 5 \wedge x - 5 < k\}$

A questo punto possiamo usare la regola dell'iterazione totale
$\vdash^{TOT}_{ITER} \{x \geq 5\} \quad P \quad \{x \geq 5 \wedge x \leq 5\}$

CORRETTEZZA TOTALE - VARIABILE NON DECREMENTA

Consideriamo il programma P

```
while x < 5 do
    x := x + 1
end-while
```

Supponiamo di voler dimostrare ? $\{x < 5\}$ P $\{x = 5\}$

Proviamo ad usare $inv = x \leq 5$ ed $E = 5 - x$, notando che $inv \rightarrow E \geq 0$.

Dobbiamo dimostrare l'invariante, quindi
? $\{x \leq 5 \land x < 5\}$ $x := x + 1$ $\{x \leq 5\}$

Usiamo la regola dell'assegnamento:
$\vdash_{ASS} \{x \leq 4\}$ $x := x + 1$ $\{x \leq 5\}$

Notiamo che $x \leq 5 \land x < 5 \equiv x \leq 4$, quindi
$\vdash \{x \leq 5 \land x < 5\}$ C $\{x \leq 5\}$, quindi l'invariante è valido.

Ora dobbiamo dimostrare la seconda premessa della regola dell'iterazione totale, ovvero
? $\{x \leq 5 \land x < 5 \land 5 - x = k\}$ $x := x + 1$ $\{x \leq 5 \land 5 - x < k\}$

Usiamo la regola dell'assegnamento:
$\vdash_{ASS} \{x \leq 4 \land 4 - x < k\}$ $x := x + 1$ $\{x \leq 5 \land 5 - x < k\}$

Notiamo che $x \leq 5 \land x < 5 \land 5 - x = k \rightarrow x \leq 4 \land 4 - x < k$, poiché se $x < 5$, allora sicuramente $x \leq 4$ perché lavoriamo con gli interi e se $5 - x = k$ allora sicuramente $4 - x < k$:
$\vdash_{IMPL} \{x \leq 5 \land x < 5 \land 5 - x = k\}$ C $\{x \leq 5 \land 5 - x < k\}$

Possiamo ora usare la regola dell'iterazione totale:
$\vdash^{TOT}_{ITER} \{x \leq 5\}$ P $\{x \leq 5 \land x \geq 5\}$

Notiamo che $x \leq 5 \land x \geq 5 \equiv x = 5$, quindi
$\vdash \{x \leq 5\}$ P $\{x = 5\}$

CORRETTEZZA TOTALE - ESPONENZIALE

Consideriamo il programma P:

```
p := 1; y := b;     //A
while y > 0 do      //W
    p := p * x; //C //W
    y := y - 1; //C //W
end-while           //W
```

Vogliamo dimostrare ? $\{b \geq 0\} \quad P \quad \{p = x^b\}$

Notiamo subito che si tratta di un assegnamento e di un ciclo, cerchiamo quindi un'invariante simulando qualche iterazione:

p	1	x	x^2	x^3	x^4
y	b	$b-1$	$b-2$	$b-3$	$b-4$

Notiamo che ad ogni passaggio $p = x^{b-y}$.

Abbiamo visto nel primo esempio di iterazione come possa capitare che un invariante semplice possa non bastare a dimostrare la validità di un ciclo, rafforziamolo da subito quindi con $y \geq 0$, che è un invariante *"garantito"* dalla condizione del ciclo.

1. DIMOSTRAZIONE INVARIANTE

Dobbiamo ora dimostrate l'invariante trovato, dobbiamo cioè dimostrare la tripla

$? \quad \{p = x^{b-y} \wedge y \geq 0 \wedge y > 0\} \quad C \quad \{p = x^{b-y} \wedge y \geq 0\}$

Notiamo subito che C consiste di due assegnamenti indipendenti, possiamo quindi usare direttamente la regola dell'assegnamento su entrambi contemporaneamente:

$\vdash_{ASS} \{px = x^{b-(y-1)} \wedge y - 1 \geq 0\} \quad C \quad \{p = x^{b-y} \wedge y \geq 0\}$

$\vdash \{p = x^{b-y} \wedge y \geq 1\} \quad C \quad \{p = x^{b-y} \wedge y \geq 0\}$

Notiamo che $p = x^{b-y} \wedge y \geq 0 \wedge y > 0 \equiv p = x^{b-y} \wedge y \geq 1$, poiché, lavorando con interi, se $y > 0$ *(che è più forte rispetto a* $y \geq 0$*)* allora sicuramente $y \geq 1$:

$\vdash \{p = x^{b-y} \wedge y \geq 0 \wedge y > 0\} \quad C \quad \{p =^{b-y} \wedge y \geq 0\}$, dunque l'invariante è valido.

Possiamo ora applicare la regola dell'iterazione parziale:

$\vdash^{PART}_{ITER} \{p = x^{b-y} \wedge y \geq 0\} \quad W \quad \{p = x^{b-y} \wedge y \geq 0 \wedge y \leq 0\}$

Notiamo che $p = x^{b-y} \wedge y \geq 0 \wedge y \leq 0 \equiv p = x^{b-y} \wedge y = 0 \equiv p = x^b$, quindi possiamo derivare

$\vdash \{p = x^{b-y} \wedge y \geq 0\} \quad W \quad \{p = x^b\}$

2. ASSEGNAMENTI INIZIALI

Adesso dobbiamo dimostrare che la premessa della tripla appena derivata sia valida dopo gli assegnamenti iniziali usando gli assegnamenti e la loro postcondizione $p = x^{b-y} \wedge y \geq 0$:

$\vdash_{ASS} \{1 = x^{b-b} \wedge b \geq 0\} \quad A \quad \{p = x^{b-y} \wedge y \geq 0\}$

$\vdash \{b \geq 0\} \quad A \quad \{p = x^{b-y} \wedge y \geq 0\}$

3. DIMOSTRAZIONE FINALE

Possiamo infine dimostrare tutto il programma utilizzando la regola della sequenza per unire le due regole ottenute, osservando che la postcondizione di una è esattamente la precondizione dell'altra:

$$\vdash_{SEQ} \{b \geq 0\} \quad P \quad \{p = x^b\}$$

4. DIMOSTRAZIONE TOTALE

Cerchiamo ora di dimostrare anche la correttezza totale del ciclo; per farlo cerchiamo una variante che diminuisca ad ogni iterazione e il cui valore ≥ 0 sia implicato dall'invariante.

Proviamo a impostare $E = y$, notando che $p = x^{b-y} \wedge y \geq 0 \rightarrow E \geq 0$:
Per dimostrare che sia valido come variante dobbiamo dimostrare la tripla

$$? \quad \{p = x^{b-y} \wedge y \geq 0 \wedge y > 0 \wedge y = k\} \quad C \quad \{p = x^{b-y} \wedge y \geq 0 \wedge y < k\}$$

Usiamo la regola dell'assegnamento:

$$\vdash_{ASS} \{px = x^{b-(y-1)} \wedge y - 1 \geq 0 \wedge y - 1 < k\} \quad C \quad \{p = x^{b-y} \wedge y \geq 0 \wedge y < k\}$$
$$\vdash \{p = x^{b-y} \wedge y \geq 1 \wedge y - 1 < k\} \quad C \quad \{p = x^{b-y} \wedge y \geq 0 \wedge y < k\}$$

Notiamo che $p = x^{b-y} \wedge y \geq 0 \wedge y > 0 \wedge y = k \rightarrow p = x^{b-y} \wedge y \geq 1 \wedge y - 1 < k$, poiché, lavorando con gli interi, se $y > 0$, allora sicuramente $y \geq 1$ e se $y = k$ allora sicuramente $y - 1 < k$:

$$\vdash_{IMPL} \{p = x^{b-y} \wedge y \geq 0 \wedge y > 0 \wedge y = k\} \quad C \quad \{p = x^{b-y} \wedge y \geq 0 \wedge y < k\}$$

Dunque il nostro variante è valido.

A questo punto possiamo applicare la regola dell'iterazione totale ottenendo:

$$\vdash^{TOT}_{ITER} \{p = x^{b-y} \wedge y \geq 0\} \quad W \quad \{p = x^{b-y} \wedge y \geq 0 \wedge y \leq 0\}$$

A questo punto dovremmo dimostrare che la precondizione di questa tripla vale dopo gli assegnamenti iniziali e ricondurci alla tripla iniziale, che è esattamente quello che abbiamo fatto prima facendo la dimostrazione parziale.

CORRETTEZZA TOTALE - FATTORIALE

Consideriamo il programma P e la tripla $?\quad \{x \geq 0\}\quad P\quad \{?\}$ *(bisogna stabilire cosa calcola il programma)*

```
z := 1;          //A
while x > 0 do       //W
    z := z * x; //C //W
    x := x - 1; //C //W
end-while            //W
```

Notiamo che il programma calcola di fatto il fattoriale di x, ma non possiamo scrivere $?\quad \{x \geq 0\}\quad P\quad \{z = x!\}$, poiché alla fine dell'esecuzione x varrà 0. Introduciamo quindi una variabile ausiliaria x_0 che *"blocchi"* il valore iniziale di x: $?\quad \{x = x_0 \wedge x \geq 0\}\quad P\quad \{z = x_0!\}$

Anche in questo caso si tratta di un ciclo e un assegnamento, quindi dobbiamo cercare un invariante. Simuliamo qualche iterazione:

z	1	x_0	$x_0(x_0-1)$	$x_0(x_0-1)(x_0-2)$
x	x_0	x_0-1	x_0-2	x_0-3

Prendiamo l'ultimo passaggio e osserviamo che $z = \frac{x_0(x_0-1)(x_0-2)(x_0-3)!}{(x_0-3)!}$
Osserviamo anche che quello che c'è sopra alla frazione non è altro che $x_0!$, mentre quello che c'è sotto è sostanzialmente x.

Possiamo dire allora che, in generale, $z = \frac{x_0!}{x!}$. Aggiungiamo all'invariante anche $x \geq 0$ per rafforzarlo *(visto che anche $x \geq 0$ è invariante)*.

1. DIMOSTRAZIONE INVARIANTE
Dobbiamo ora dimostrare la validità dell'invariante proposto, ovvero dimostrare la tripla $?\quad \{z = \frac{x_0!}{x!} \wedge x \geq 0 \wedge x > 0\}\quad C\quad \{z = \frac{x_0!}{x!} \wedge x \geq 0\}$

Usiamo la regola dell'assegnamento, ma stiamo attenti: ora gli assegnamenti non sono più indipendenti, quindi dobbiamo gestirli separatamente:

1. $\vdash_{ASS} \{z = \frac{x_0!}{(x-1)!} \wedge x - 1 \geq 0\}\quad x := x - 1\quad \{z = \frac{x_0!}{x!} \wedge x \geq 0\}$
2. $\vdash_{ASS} \{zx = \frac{x_0!}{(x-1)!} \wedge x - 1 \geq 0\}\quad z := z * x\quad \{z = \frac{x_0!}{(x-1)!} \wedge x - 1 \geq 0\}$
   $\vdash \{z = \frac{x_0!}{x!} \wedge x - 1 \geq 0\}\quad z := z * x\quad \{z = \frac{x_0!}{(x-1)!} \wedge x - 1 \geq 0\}$

A questo punto usiamo la regola della sequenza per unire le due triple, notando che la precondizione della prima è la postcondizione della seconda:
$\vdash_{SEQ} \{z = \frac{x_0!}{x!} \wedge x - 1 \geq 0\}\quad C\quad \{z = \frac{x_0!}{x!} \wedge x \geq 0\}$

Notiamo che $z = \frac{x_0!}{x!} \wedge x \geq 0 \wedge x > 0 \equiv z = \frac{x_0!}{x!} \wedge x - 1 \geq 0$, quindi:
$\vdash \{z = \frac{x_0!}{x!} \wedge x \geq 0 \wedge x > 0\}\quad C\quad \{z = \frac{x_0!}{x!} \wedge x \geq 0\}$
E quindi l'invariante proposto è valido.

Possiamo ora usare la regola dell'iterazione parziale:
$\vdash^{PART}_{ITER} \{z = \frac{x_0!}{x!} \wedge x \geq 0\}\quad W\quad \{z = \frac{x_0!}{x!} \wedge x \geq 0 \wedge x \leq 0\}$

Notiamo adesso che $z = \frac{x_0!}{x!} \wedge x \geq 0 \wedge x \leq 0 \equiv z = \frac{x_0!}{x!} \wedge x = 0 \equiv z = x0!$:
$\vdash \{z = \frac{x_0!}{x!} \wedge x \geq 0\}\quad W\quad \{z = x_0!\}$

2. ASSEGNAMENTO INIZIALE

Ora dobbiamo dimostrare che la precondizione della tripla ottenuta sia valida dopo l'assegnamento iniziale:

$\vdash_{ASS} \{1 = \frac{x_0!}{x!} \land x \geq 0\} \quad A \quad \{z = \frac{x_0!}{x!} \land x \geq 0\}$

$\vdash \{x = x_0 \land x \geq 0\} \quad A \quad \{z = \frac{x_0!}{x!} \land x \geq 0\}$

3. DIMOSTRAZIONE FINALE

A questo punto possiamo unire le triple derivate con la regola della sequenza e concludere la dimostrazione:

$\vdash_{SEQ} \{x = x_0 \land x \geq 0\} \quad P \quad \{z = x_0!\}$

4. DIMOSTRAZIONE TOTALE

Cerchiamo ora di dimostrare anche la correttezza totale del ciclo; per farlo cerchiamo una variante che diminuisca ad ogni iterazione e il cui valore ≥ 0 sia implicato dall'invariante.

Proviamo ad impostare $E = x$, osservando che $z = \frac{x_0!}{x!} \land x \geq 0 \rightarrow E = x \geq 0$.
Per dimostrare la validità del variante dobbiamo dimostrare la tripla

$? \quad \{z = \frac{x_0!}{x!} \land x \geq 0 \land x > 0 \land x = k\} \quad C \quad \{z = \frac{x_0!}{x!} \land x \geq 0 \land x < k\}$

Usiamo quindi la regola dell'assegnamento:

1. $\vdash_{ASS} \{z = \frac{x_0!}{(x-1)!} \land x - 1 \geq 0 \land x - 1 < k\} \quad x := x - 1 \quad \{z = \frac{x_0!}{x!} \land x \geq 0 \land x < k\}$
2. $\vdash_{ASS} \{zx = \frac{x_0!}{(x-1)!} \land x - 1 \geq 0 \land x - 1 < k\} \quad z := z * x \quad \{z = \frac{x_0!}{(x-1)!} \land x - 1 \geq 0 \land x - 1 < k\}$

   $\vdash \{z = \frac{x_0!}{x!} \land x - 1 \geq 0 \land x - 1 < k\} \quad z := z * x \quad \{z = \frac{x_0!}{(x-1)!} \land x - 1 \geq 0 \land x - 1 < k\}$

Usiamo ora la regola della sequenza:

$\vdash_{SEQ} \{z = \frac{x_0!}{x!} \land x \geq 1 \land x - 1 < k\} \quad C \quad \{z = \frac{x_0!}{x!} \land x \geq 0 \land x < k\}$

Notiamo ora che $z = \frac{x_0!}{x!} \land x \geq 0 \land x > 0 \land x = k \rightarrow z = \frac{x_0!}{x!} \land x \geq 1 \land x - 1 < k$, poiché se $x > 0$, allora sicuramente $x \geq 1$ e se $x = k$ sicuramente $x - 1 < k$:

$\vdash_{IMPL} \{z = \frac{x_0!}{x!} \land x \geq 0 \land x > 0 \land x = k\} \quad C \quad \{z = \frac{x_0!}{x!} \land x \geq 0 \land x < k\}$

Da qui possiamo applicare la regola dell'iterazione totale:

$\vdash^{TOT}_{ITER} \{z = \frac{x_0!}{x!} \land x \geq 0\} \quad W \quad \{z = \frac{x_0!}{x!} \land x \geq 0 \land x \leq 0\}$

Infine per terminare la dimostrazione dovremmo mostrare che la precondizione della tripla ottenuta è valida anche dopo il primo assegnamento e concatenare il tutto come fatto prima.

CORRETTEZZA TOTALE - ESEMPIO COMPLESSO: MOLTIPLICAZIONE RUSSA

Consideriamo il programma P

```
x := m; y := n; z := 0;     //A
while x != 0 do                     //W
    if( x mod 2 = 0 ) then      //C //W
        x := x / 2;         //Y //C //W
        y = y * 2;          //Y //C //W
    else                        //C //W
        x := x - 1;         //N //C //W
        z := z + y;         //N //C //W
    end-if                      //C //W
end-while                           //W
```

Vogliamo dimostrare ? $\{m > 0 \land n > 0\} \quad P \quad \{z = mn\}$

Cerchiamo innanzitutto un invariante:
Notiamo che nel primo ramo dell'`if` vale la relazione invariante $xy = k$.
Nel secondo ramo invece abbiamo $k = (x-1)y \implies k - y$ e $z = z + y$, dunque in questo ramo abbiamo una quantità invariante $xy + z$.

A questo punto notiamo che nel primo ramo z non cambia, dunque possiamo dire che $k = xy + z$ è un invariante per il ciclo.

Ma cos'è k?
Osserviamo che dopo gli assegnamenti iniziali $xy + z = mn$, quindi $k = mn$; l'invariante è dunque $mn = xy + z$.

1. DIMOSTRAZIONE INVARIANTE
Dobbiamo ora dimostrare l'invariante, ovvero
$? \quad \{mn = xy + z \land x \neq 0\} \quad C \quad \{mn = xy + z\}$

Notiamo che C corrisponde ad una scelta, dobbiamo quindi dimostrare le due premesse della regola della scelta prima, ovvero:
$? \quad \{mn = xy + z \land x \neq 0 \land x \mod 2 = 0\} \quad Y \quad \{mn = xy + z\}$
$? \quad \{mn = xy + z \land x \neq 0 \land x \mod 2 \neq 0\} \quad N \quad \{mn = xy + z\}$

Partiamo dal primo ramo, notando che si tratta di due assegnamenti indipendenti:
$\vdash_{ASS} \{mn = (x/2)(2y) + z\} \quad Y \quad \{mn = xy + z\}$
$\vdash \{mn = xy + z\} \quad Y \quad \{mn = xy + z\}$

Notiamo che $mn = xy + z \land x \neq 0 \land x \mod 2 = 0 \to mn = xy + z$:
$\vdash_{IMPL} \{mn = xy + z \land x \neq 0 \land x \mod 2 = 0\} \quad Y \quad \{mn = xy + z\}$

Passiamo al secondo ramo; anche qui abbiamo due assegnamenti indipendenti:
$\vdash_{ASS} \{mn = (x-1)y + z + y\} \quad N \quad \{mn = xy + z\}$
$\vdash \{mn = xy - y + z + y\} \quad N \quad \{mn = xy + z\}$
$\vdash \{mn = xy + z\} \quad N \quad \{mn = xy + z\}$

Notiamo anche qui che $mn = xy + z \land x \neq 0 \land x \mod 2 \neq 0 \to mn = xy + z$:
$\vdash_{IMPL} \{mn = xy + z \land x \neq 0 \land x \mod 2 \neq 0\} \quad N \quad \{mn = xy + z\}$

A questo punto possiamo usare la regola della scelta:
$\vdash_{IF} \{mn = xy + z \land x \neq 0\} \quad C \quad \{mn = xy + z\}$
Dunque l'invariante è valido.

Possiamo ora usare la regola dell'iterazione parziale:
$\vdash^{PART}_{ITER} \{mn = xy + z\} \quad W \quad \{mn = xy + z \land x = 0\}$

Notiamo intanto che $mn = xy + z \land x = 0 \equiv mn = z$, per cui:
$\vdash \{mn = xy + z\} \quad W \quad \{mn = z\}$

2. ASSEGNAMENTI INIZIALI
Dobbiamo ora dimostrare che la precondizione del ciclo vale dopo gli assegnamenti iniziali, cioè
$? \quad \{x > 0 \land y > 0\} \quad A \quad \{mn = xy + z\}$

Notiamo che si tratta di assegnamenti indipendenti, per cui:
$\vdash_{ASS} \{mn = mn + 0\} \quad A \quad \{mn = xy + z\}$, ovvero
$\vdash \{True\} \quad A \quad \{mn = xy + z\}$

Sappiamo che $True$ è implicato da qualsiasi formula, quindi:
$\vdash_{IMPL} \{x > 0 \land y > 0\} \quad A \quad \{mn = xy + z\}$

3. DIMOSTRAZIONE FINALE
Possiamo ora usare la regola della sequenza per unire le triple degli assegnamenti iniziali e del ciclo e completare la dimostrazione parziale:
$\vdash_{SEQ} \quad \{x > 0 \land y > 0\} \quad P \quad \{mn = z\}$

4. DIMOSTRAZIONE TOTALE
Per fare la dimostrazione totale ci serve un variante di ciclo E il cui valore ≥ 0 sia implicato da un invariante.
Usiamo un altro invariante, ovvero $mn = xy + z \land x \geq 0$.

Dimostriamo questo invariante, cioè
$? \quad \{mn = xy + z \land x \geq 0 \land x \neq 0\} \quad C \quad \{mn = xy + z \land x \geq 0\}$

Deriviamo innanzitutto le premesse della regola della scelta
$? \quad \{mn = xy + z \land x \geq 0 \land x \neq 0 \land x \mod 2 = 0\} \quad Y \quad \{mn = xy + z \land x \geq 0\}$
$? \quad \{mn = xy + z \land x \geq 0 \land x \neq 0 \land x \mod 2 \neq 0\} \quad N \quad \{mn = xy + z \land x \geq 0\}$

$\vdash_{ASS} \{mn = (x/2)(2y) + z \land (x/2) \geq 0\} \quad Y \quad \{mn = xy + z \land x \geq 0\}$
$\vdash \{mn = xy + z \land x \geq 0\} \quad Y \quad \{mn = xy + z \land x \geq 0\}$
$\vdash_{IMPL} \{mn = xy + z \land x \geq 0 \land x \neq 0 \land x \mod 2 = 0\} \quad Y \quad \{mn = xy + z \land x \geq 0\}$

$\vdash_{ASS} \{mn = (x-1)y + z + y \land x - 1 \geq 0\} \quad N \quad \{mn = xy + z \land x \geq 0\}$
$\vdash \{mn = xy - y + z + y \land x \geq 1\} \quad N \quad \{mn = xy + z \land x \geq 0\}$
$\vdash \{mn = xy + z \land x \geq 1\} \quad N \quad \{mn = xy + z \land x \geq 0\}$

Notiamo che $x \geq 0 \land x \neq 0 \equiv x > 0 \equiv x \geq 1$, quindi:
$\vdash \{mn = xy + z \land x \geq 0 \land x \neq 0 \land x \mod 2 \neq 0\} \quad N \quad \{mn = xy + z \land x \geq 0\}$

$\vdash_{IF} \{mn = xy + z \land x \geq 0 \land x \neq 0\} \quad C \quad \{mn = xy + z \land x \geq 0\}$
Dunque anche questo invariante è valido.

Notiamo che $mn = xy + z \land x \geq 0 \rightarrow x \geq 0$, quindi proviamo ad impostare $E = x$.
Per dimostrare che il variante è valido dobbiamo dimostrare
$? \quad \{mn = xy + z \land x \geq 0 \land x \neq 0 \land x = k\} \quad C \quad \{mn = xy + z \land x \geq 0 \land x < k\}$

Anche qui, dobbiamo dimostrare le precondizioni della regola di scelta:
$? \quad \{mn = xy + z \land x \geq 0 \land x \neq 0 \land x = k \land x \mod 2 = 0\} \quad Y \quad \{inv \land x < k\}$
$? \quad \{mn = xy + z \land x \geq 0 \land x \neq 0 \land x = k \land x \mod 2 \neq 0\} \quad N \quad \{inv \land x < k\}$

$\vdash_{ASS} \{mn = (x/2)(2y) + z \land (x/2) \geq 0 \land (x/2) < k\} \quad Y \quad \{mn = xy + z \land x \geq 0 \land x < k\}$

$\vdash \{mn = xy + z \land x \geq 0 \land x < 2k\} \quad Y \quad \{mn = xy + z \land x \geq 0 \land x < k\}$
Notiamo che $x = k \rightarrow x < 2k$, quindi
$\vdash_{IMPL} \{mn = xy + z \land x \geq 0 \land x \neq 0 \land x = k \land x \mod 2 = 0\} \quad Y \quad \{inv \land x < k\}$

$\vdash_{ASS} \{mn = (x-1)y + z + y \land x - 1 \geq 0 \land x - 1 < k\} \quad N \quad \{mn = xy + z \land x \geq 0 \land x < k\}$

$\vdash \{mn = xy + z \land x \geq 1 \land x - 1 < k\} \quad N \quad \{mn = xy + z \land x \geq 0 \land x < k\}$
Notiamo che $x \geq 0 \land x \neq 0 \equiv x \geq 1$ e che $x = k \rightarrow x - 1 < k$
$\vdash_{IMPL} \{mn = xy + z \land x \geq 0 \land x \neq 0 \land x = k \land x \mod 2 \neq 0\} \quad N \quad \{inv \land x < k\}$

Possiamo ora usare la regola della scelta:
$\vdash_{IF} \{mn = xy + z \land x \geq 0 \land x = k\} \quad C \quad \{mn = xy + z \land x \geq 0 \land x < k\}$

Notiamo ora che $x \geq 0 \land x \neq 0 \rightarrow x \geq 0$, quindi:
$\vdash_{IMPL} \{mn = xy + z \land x \geq 0 \land x \neq 0 \land x = k\} \quad C \quad \{mn = xy + z \land x \geq 0 \land x < k\}$

Abbiamo dimostrato che il variante è valido. Possiamo ora usare la regola dell'iterazione totale:
$\vdash^{TOT}_{ITER} \{mn = xy + z \land x \geq 0\} \quad W \quad \{mn = xy + z \land x \geq 0 \land x = 0\}$

Notiamo che $x \geq 0 \land x = 0 \rightarrow x = 0$ e che $x = 0 \rightarrow mn = z$, quindi
$\vdash_{IMPL} \{mn = xy + z \land x \geq 0\} \quad W \quad \{mn = z\}$

Ora dobbiamo dimostrare che la precondizione della tripla appena derivata valga anche dopo gli assegnamenti iniziali, cioè $? \quad \{m > 0 \land n > 0\} \quad A \quad \{mn = xy + z \land x \geq 0\}$

$\vdash_{ASS} \{mn = mn + 0 \land m \geq 0\} \quad A \quad \{mn = xy + z \land x \geq 0\}$
$\vdash \{m \geq 0\} \quad A \quad \{mn = xy + z \geq 0 \land x \geq 0\}$

Notiamo che $m > 0 \land n > 0 \rightarrow m \geq 0$, quindi
$\vdash_{IMPL} \{m > 0 \land n > 0\} \quad A \quad \{mn = xy + z \geq 0 \land x \geq 0\}$

A questo punto possiamo terminare la dimostrazione usando la regola della sequenza:
$\vdash_{SEQ} \{m > 0 \land n > 0\} \quad P \quad \{mn = z\}$

CORRETTEZZA TOTALE - QUADRATO

Consideriamo il programma P:

```
n := 1; s := 0; k := 0;      //A
while k < p do               //W
    s := s + n;      //C1     //W
    n := n + 2;      //C2     //W
    k := k + 1;      //C3     //W
end-while                    //W
```

Vogliamo derivare ? $\{p \geq 0\} \quad P \quad \{s = p^2\}$

Il programma sfrutta una proprietà che dice che se sommiamo i primi k numeri dispari, otteniamo k^2.

Cerchiamo quindi un'invariante opportuno. Notiamo subito che $k \leq p$ è invariante per W.
Simuliamo qualche iterazione:

s	0	1	4	9	16
n	1	3	5	7	9
k	0	1	2	3	4

Notiamo che un'altra relazione invariante è $s = k^2$, ma anche $n = 2k + 1$.

1. DIMOSTRAZIONE INVARIANTE

Il candidato invariante dunque è $inv \implies s = k^2 \wedge n = 2k + 1 \wedge k \leq p$; per dimostrarlo dobbiamo derivare la tripla ? $\{inv \wedge k < p\} \quad C \quad \{inv\}$.

Notiamo subito che C è composto da assegnamenti non indipendenti, quindi dobbiamo trattare ogni assegnamento in disparte, partendo dal fondo *(poiché disponiamo di una precondizione)*.

$\vdash_{ASS} \{s = (k+1)^2 \wedge n = 2(k+1) + 1 \wedge k + 1 \leq p\} \quad C_3 \quad \{inv\}$
$\vdash_{ASS} \{s = (k+1)^2 \wedge n + 2 = 2(k+1) + 1 \wedge k + 1 \leq p\} \quad C_2 \quad \{pre\ C_3\}$
$\vdash_{ASS} \{s + n = (k+1)^2 \wedge n + 2 = 2(k+1) + 1 \wedge k + 1 \leq p\} \quad C_1 \quad \{pre\ C_2\}$

$\vdash \{s + n = k^2 + 2k + 1 \wedge n + 2 = 2k + 2 + 1 \wedge k + 1 \leq p\} \quad C_1 \quad \{pre\ C_2\}$
$\vdash \{s + n = k^2 + n \wedge n = 2k + 1 \wedge k + 1 \leq p\} \quad C_1 \quad \{pre\ C_2\}$
$\vdash \{s = k^2 \wedge n = 2k + 1 \wedge k + 1 \leq p\} \quad C_1 \quad \{pre\ C_2\}$

Notiamo che $k + 1 \leq p \to k < p$, quindi
$\vdash_{IMPL} \{s = k^2 \wedge n = 2k + 1 \wedge k < p\} \quad C_1 \quad \{pre\ C_2\}$

A questo punto osserviamo che $inv \wedge k < p \equiv s = k^2 \wedge n = 2k + 1 \wedge k < p$, poiché $k < p$ è più forte di $k \leq p$;
$\vdash \{inv \wedge k < p\} \quad C_1 \quad \{pre\ C_2\}$

Usiamo adesso due volte la regola della sequenza:
$\vdash_{SEQ} \{inv \wedge k < p\} \quad C_1; C_2 \quad \{pre\ C_3\}$
$\vdash_{SEQ} \{inv \wedge k < p\} \quad C \quad \{inv\}$

Abbiamo dimostrato la validità dell'invariante.
Ora possiamo usare la regola dell'iterazione parziale:
$\vdash^{PART}_{ITER} \{inv\} \quad W \quad \{inv \wedge k \geq p\}$

2. ASSEGNAMENTI INIZIALI

Dobbiamo ora dimostrare che l'invariante è valido dopo gli assegnamenti iniziali, cioè
$? \quad \{p \geq 0\} \quad A \quad \{inv\}$

$\vdash_{ASS} \{s = 0^2 \wedge 1 = 2 * 0 + 1 \wedge 0 \leq p\} \quad A \quad \{inv\}$
$\vdash \{s = 0 \wedge p \geq 0\} \quad A \quad \{inv\}$

Notiamo che $s = 0 \wedge p \geq 0 \rightarrow p \geq 0$, per cui
$\vdash_{IMPL} \{p \geq 0\} \quad A \quad \{inv\}$

3. DIMOSTRAZIONE FINALE

Possiamo ora unire le due regole derivate con la regola della sequenza
$\vdash_{SEQ} \{p \geq 0\} \quad P \quad \{s = k^2 \wedge n = 2k + 1 \wedge k < p \wedge k \geq p\}$

Notiamo che $k < p \wedge k \geq p \equiv k = p$
$\vdash \{p \geq 0\} \quad P \quad \{s = p^2 \wedge n = 2p + 1 \wedge k = p\}$

Osserviamo che $s = p^2 \wedge n = 2p + 1 \wedge k = p \rightarrow s = p^2$
$\vdash_{IMPL} \{p \geq 0\} \quad P \quad \{s = p^2\}$

Abbiamo terminato la dimostrazione parziale del programma P.

3. CORRETTEZZA TOTALE

Per dimostrare la correttezza totale del programma abbiamo bisogno di un *variante* E tale per cui $inv \rightarrow E \geq 0$ e che diminuisca ad ogni iterazione.

Notiamo che ad ogni iterazione k viene incrementato di 1; proviamo ad impostare $E = p - k$, notando che l'invariante impone $k \leq p$, per cui $inv \rightarrow E \geq 0$.

Per verificare l'invariante dobbiamo dimostrare $? \quad \{inv \wedge k < p \wedge E = c\} \quad C \quad \{inv \wedge E < c\}$

$\vdash_{ASS} \{s = (k+1)^2 \wedge n = 2(k+1) + 1 \wedge k + 1 \leq p \wedge p - (k+1) < c\} \quad C_3 \quad \{inv \wedge p - k < c\}$

$\vdash_{ASS} \{s = (k+1)^2 \wedge n + 2 = 2(k+1) + 1 \wedge k + 1 \leq p \wedge p - (k+1) < c\} \quad C_2 \quad \{pre\ C_3\}$
$\vdash_{ASS} \{s + n = (k+1)^2 \wedge n + 2 = 2(k+1) + 1 \wedge k + 1 \leq p \wedge p - (k+1) < c\} \quad C_1 \quad \{pre\ C_2\}$

$\vdash \{s = k^2 \wedge n = 2k + 1 \wedge k + 1 \leq p \wedge p - (k+1) < c\} \quad C_1 \quad \{pre\ C_2\}$

$\vdash \{inv \wedge k < p \wedge p - (k+1) < c\} \quad C_1 \quad \{pre\ C_2\}$
$\vdash_{SEQ} \{inv \wedge k < p \wedge p - (k+1) < c\} \quad C \quad \{inv \wedge p - k < c\}$

Notiamo che $p - k = c \rightarrow p - k - 1 < c$, dunque
$\vdash_{IMPL} \{inv \wedge k < p \wedge p - k = c\} \quad C \quad \{inv \wedge p - k < c\}$

Abbiamo dimostrato la validità del variante, possiamo ora usare la regola dell'iterazione completa:
$\vdash^{TOT}_{ITER} \{inv\} \quad W \quad \{inv \wedge k \geq p\}$

Dobbiamo poi dimostrare che l'invariante è valido dopo l'assegnamento iniziale, come fatto prima, e abbiamo terminato la dimostrazione.

MODELLI DI SISTEMI CONCORRENTI

Abbiamo visto come funzionano le triple di Hoare e come è possibile sfruttarle per calcolare la correttezza dei programmi sequenziali.
Inoltre, scrivere $\{p\} \quad S \quad \{q\}$ equivale anche di specificare il significato del programma S. Questo tipo di ragionamento che permette di specificare S o di avere delle tecniche per provare la correttezza di S viene detto ***semantica assiomatica***, ovvero un modo per dare semantica, significato al programma S attraverso gli assiomi della logica di Hoare.

Nel caso della programmazione sequenziale esistono anche altri modi per dare una semantica; possiamo ad esempio vedere il programma S come una funzione che trasforma i dati in input in dati in output $F_S : I \to O$; questa prende il nome di ***semantica denotazionale***. A tal proposito esiste anche il **lambda calcolo** che permette di associare delle funzioni ai programmi ed è quindi un calcolo che permette di combinare funzioni e che è stato studiato quali sono le funzioni computabili e quali sono incomputabili.

Un altro modo ancora per dare una semantica ai programmi sequenziale è la ***semantica operazionale*** che consiste nel prendere una macchina astratta facendo vedere come simulare un programma su di esse *(ad esempio la macchina di Turing)*.

PROGRAMMI CONCORRENTI
Tutto ciò appena detto è valido per i programmi sequenziali, ma cosa succede nel caso dei programmi concorrenti?
Ci sono alcune diversità: ad esempio, nel caso della programmazione sequenziale, è fondamentale che il programma termini, mentre nel caso della programmazione concorrente spesso è necessario che i programmi non termino e siano sempre attivi.

Ci sono però anche altre esigenze. Facciamo un esempio:
Supponiamo di avere due programmi con le relative triple di Hoare:
$S_1 : \quad x = 2 \qquad\qquad \{x = v\} \quad S_1 \quad \{x = 2\}$
$S_2 : \quad x = 3 \qquad\qquad \{x = v\} \quad S_1 \quad \{x = 3\}$

Notiamo che possiamo eseguire i due programmi in sequenza $S_1; S_2$; in questo caso avremo $\{x = v\} \quad S_1; S_2 \quad \{x = 3\}$.

Se prendiamo $S_1' : \{x = v\} \quad x = 1; x = x + 1 \quad \{x = 2\}$; notiamo che S_1 e S_1' soddisfano la stessa specifica, quindi se sostituiamo S_1' ad S_1 otterremmo la stessa tripla $\{x = v\} \quad S_1'; S_2 \quad \{x = 3\}$. Si parla di **composizionalità**; se sostituiamo un programma con uno che calcola la stessa funzione in una composizione di programmi, la composizione non cambia.

Vediamo ora però cosa succede se anziché comporre i programmi in sequenza li componiamo in parallelo $S_1|S_2$; avremo $\{x = v\} \quad S_1|S_2 \quad \{x = 2 \vee x = 3\}$. Già qui abbiamo del **non determinismo**.

Cosa succede in questo caso se sostituiamo S_1' ad S_1?
$\{x = v\} \quad S_1'|S_2 \quad \{x = 2 \vee x = 3 \vee x = 4\}$ perché potrei ad esempio eseguire prima $x = 1$, poi $x = 3$ e infine $x = x + 1$. Abbiamo quindi **perso** la composizionalità in questo caso.

CALCULUS COMMUNICATING SYSTEMS

Abbiamo visto i problemi riscontrati nel passaggio da programmi concorrenti a programmi sequenziali; due ricercatori stavano lavorando su questi problemi, ovvero **R. Milner** e **C.A.R. Hoare**.
In questo periodo *(anni '70 - '80)* Milner introduce il **Calculus Communicating Systems *(CCS)*** mentre Hoare introduce i **Comunicating Sequential Processes *(CSP)***.

Il CSP è un nucleo di linguaggi di programmazione pensato per la programmazione concorrente, mentre il CCS è un calcolo che vuole generalizzare il lambda calcolo al caso concorrente.
L'idea sottostante è la stessa: un **sistema** è fatto di **processi** ciascuno con una memoria privata e un comportamento indipendente che interagiscono tra loro scambiandosi **messaggi** e con l'ambiente. Inoltre, se una variabile viene condivisa fra più processi, diventa essa stessa un processo.
La sincronizzazione avviene a due a due: un processo esegue l'azione a e un altro esegue la coazione $\overline{a}$.
I sistemi di questo tipo vengono chiamati **sistemi reattivi** *(reagiscono all'ambiente)*.

Milner cercò di reintrodurre la composizionalità definendo l'**equivalenza all'osservazione**: prima due programmi erano equivalenti se calcolavano la stessa funzione *(soddisfavano la stessa tripla di Hoare)*, ora due sistemi sono equivalenti quando, sostituendo un processo, né l'ambiente né gli altri processi si accorgono di tale sostituzione. Vedremo che a volte non basta nemmeno che due processi eseguano le stesse sequenze di operazioni per essere equivalenti all'osservazione.

Noi vedremo la nozione di equivalenza all'osservazione che prende il nome di **bisimulazione** che è la più conosciuta e la più utilizzata, ma è anche molto restrittiva.

LABELLED TRANSITION SYSTEMS (LTS)

Per dare la semantica del CCS ci sono vari modi; noi considereremo i **sistemi di transizione etichettati** che sono dei grafi di transizione dove i nodi rappresentano i processi che devono ancora essere eseguiti, la transizione rappresenta l'azione da eseguire *(con cui è anche etichettata)* e il nodo successivo rappresenta il nuovo processo da eseguire dopo l'esecuzione della transizione.

Un altro modo per dare una semantica a questi sistemi sono le **strutture di Kripke** che sono sempre dei grafi dove però i nodi rappresentano gli stati e sono etichettati con le proprietà vere in quello stato mentre le transizioni fanno passare da uno stato all'altro *(li vedremo in seguito)*.

Quindi un LTS non è altro che un automa a stati finiti ed è formato come segue:
$LTS(S, Act, T, s_0)$ con

- S: insieme di stati
- Act: insieme di nomi di azioni elementari
- $T = \{(s, a, s') \mid s, s' \in S,\ a \in Act\}$: insieme di transizioni
  $T \subseteq S \times Act \times S$
  In particolare, se $(s, a, s') \in T$, possiamo rappresentarlo come $s \rightarrow^a s'$, oppure come due nodi s ed s' collegati da un arco etichettato con a.
- s_0: stato iniziale *(non sempre presente)*

Possiamo estendere le transizioni $s \to^a s'$ alle sequenze $w \in Act^*$:
$s \to^w s'$ se e solo se:

- se $|w| = 0$, ovvero se $w = \epsilon$ e in questo caso specifico $s = s'$
- se $|w| > 0$, possiamo scomporre w in a concatenato a x $w = a \cdot x$ con $a \in Act$ e $x \in Act^*$
  Inoltre se $s \to^a s_1 \to^x s'$, allora $s \to^{ax} s'$.

NOTAZIONI

- **RIPASSO CLASSI DI EQUIVALENZA**

  Diciamo $R \subseteq X \times X$ R relazione binaria su X
  R è riflessiva $\forall x \in X \ : \ x \ R \ x$
  R è simmetrica $\forall x, y \in X$ se $x \ R \ y$, allora $y \ R \ x$
  R è transitiva $\forall x, y, z \in X$ se $x \ R \ y$ e $y \ R \ z$ allora $x \ R \ z$

  Se R è riflessiva, simmetrica e transitiva allora R è una **relazione d'equivalenza**.
  Questo significa che prendendo un elemento di X possiamo costruire la sua classe di equivalenza $[x] = \{y \in X \mid x \ R \ y\}$.
  Il concetto fondamentale è che se R è una relazione di equivalenza, allora l'insieme X viene ripartito in classi di equivalenza **disgiunte**.

  Se prendiamo $[x_1]$ e $[x_2]$ due classi di equivalenza degli elementi x_1 e x_2 con $x_1 \neq x_2$, avremmo due possibili casi:

  1. $[x_1] = [x_2]$ cioè x_1 e x_2 sono due rappresentanti diversi della stessa classe
  2. $[x_1] \cap [x_2] = \emptyset$ e quindi sono classi diverse

  Inoltre $\bigcup_{x \in X} [x] = X$

- **RIPASSO RELAZIONI**
  Prendiamo $R, \ R', \ R'' \subseteq X \times X$
  Diremo che R' è la ***chiusura riflessiva/simmetrica/transitiva*** di R sse $R \subseteq R'$, ovvero R' è la più piccola relazione che contiene R tale che R' è ***riflessiva/simmetrica/transitiva***.

- Useremo $s \to s'$ se $\exists \ a \in Act : s \to^a s'$ $\to \ = \bigcup_{a \in Act} \to^a$ e $\to \subseteq S \times S$

- Useremo $s \to^* s'$ se $\exists \ w \in Act^* : s \to^w s'$ $\to^* \ = \bigcup_{w \in Act^*} \to^w$ e $\to^* \subseteq S \times S$

  $\to^*$ è la chiusura transitiva riflessiva di $\to$

CCS PURO

Diamo ora qualche definizione dei termini del sistema di transizione.

- K insieme di nomi di processi *(es. Buffer, Sender, p, q)*
- A insieme di nomi di azioni
- $\overline{A}$ insieme di nomi di coazione $\quad \forall\, a \in A \;\; \exists\, \overline{a} \in \overline{A} \qquad \overline{\overline{a}} = a \;\; \forall\, a \in A$
- $Act = A \cup \overline{A} \cup \{\mathcal{T}\} \qquad \mathcal{T} \notin A \qquad \mathcal{T} \notin \overline{A}$
  Diremo che A e $\overline{A}$ sono le **azioni osservabili**, ovvero sono sincronizzazioni con l'ambiente del processo considerato, mentre $\mathcal{T}$ sono il risultato di una sincronizzazione e quindi sono **azioni non osservabili** o ***sincronizzazioni interne*** *(è l'ambiente che osserva)*.

Possiamo quindi dire che un processo CCS è un'espressione che utilizza i simboli di azione e che è composta con gli operatori del CCS.
Un sistema CCS è un insieme di processi $p \in K$.

Avremo quindi $p =$ espressione CCS.
Avendo un insieme di processi, avremo un sistema di equazioni CCS.

OPERATORI CCS

- $Nil \in P_{CSS}$ è il processo che non fa niente e viene rappresentato come uno stato Nil che non ha transizioni uscenti.
- Operatore *prefisso*: se $p \in P_{CCS}$ e $\alpha \in Act$, allora $\alpha \cdot p \in P_{CCS}$
  Praticamente il processo $\alpha \cdot p$ può eseguire α e dopo si comporterà come p.

  Regola di inferenza: $\frac{}{\alpha \cdot p \to^{\alpha} p}$
- Operatore *selezione*: se $p_1,\ p_2 \in P_{CCS}$ allora $p_1 + p_2$ è il processo che si comporta o come p_1 o come p_2.

  Regola di inferenza: $\frac{p_1 \to^{\alpha} p_1'}{p1 + p2 \to^{\alpha} p_1'} \quad \vee \; \frac{p_2 \to^{\beta} p_2'}{p1 + p2 \to^{\beta} p_2'}$
- Operatore *composizione parallela*: se $p_1,\ p_2 \in P_{CCS}$ allora $p_1 \mid p_2 \in P_{CCS}$.

  Regola di inferenza: $\frac{p_1 \to^{\alpha} p_1'}{p1 \mid p2 \to^{\alpha} p_1'} \qquad \frac{p_2 \to^{\beta} p_2'}{p1 \mid p2 \to^{\beta} p_2'} \qquad \frac{p_1 \to^{\alpha} p_1' \wedge p_2 \to^{\overline{\alpha}} p_2'}{p1 \mid p2 \to^{\mathcal{T}} p_1' \mid p_2'}$
  Quindi può andare avanti un processo piuttosto che un altro oppure può esserci una sincronizzazione interna tra i processi in parallelo.
- Operatore *restrizione*: se $L \subseteq A$ e $p \in P_{CCS}$ allora $P \setminus L$ è il processo che non può eseguire le azioni dell'insieme $L \cup \overline{L}$ se non quando sono delle sincronizzazioni. Ovvero ci dice che le azioni L sono locali e quindi il sistema non interagisce con l'ambiente con le azioni di L.

  Senza la restrizione un processo potrebbe eseguire prima l'azione e poi la coazione o viceversa.

  Regola di inferenza: $\frac{p \to^{\alpha} p' \wedge \alpha, \overline{\alpha} \notin L}{p \setminus L \to^{\alpha} p' \setminus L}$
- Operatore *rietichettatura*: abbiamo una funzione $f : Act \to Act$ tale che $f(\mathcal{T}) \to \mathcal{T}$ e $f(\overline{a}) \to \overline{f(a)}$
  Ovvero la funzione non può rinominare le $\mathcal{T}$ e deve mantenere la relazione che c'è tra azioni e coazioni.

  Regola di inferenza: $\frac{p \to^{\alpha} p'}{p[f] \to^{f(\alpha)} p'[f]}$

Gli operatori CCS rispettano la seguente precedenza: $\setminus L,\ [f],\ \alpha \cdot p,\ |,\ +$

ESEMPIO CCS

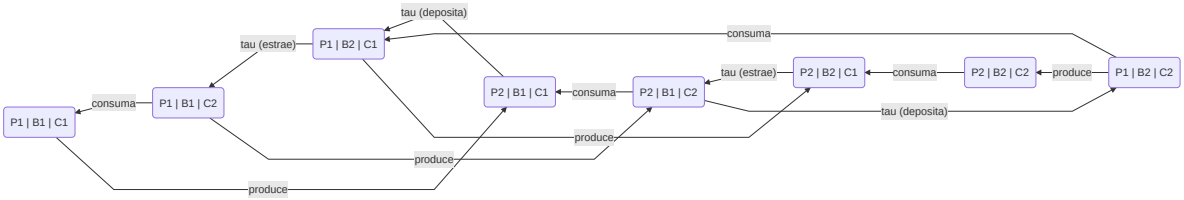
$S = (P_1 \mid B_1 \mid C_1) \setminus \{deposita,\ estrae\}$ e quindi
$K = \{S,\ P_1,\ B_1,\ C_1,\ P_2,\ B_2,\ C_2\}$
$A = \{deposita,\ estrae,\ produce,\ consuma\}$
$\overline{A} = \{\overline{deposita},\ \overline{estrae},\ \overline{produce},\ \overline{consuma}\}$

$\setminus$ forza i processi P_1, B_1 e C_1 a sincronizzarsi internamente con le azioni dopo il simbolo, evitando quindi che quelle azioni si sincronizzino con l'ambiente.

$P_1 = produce \cdot P_2$
$P_2 = \overline{deposita} \cdot P_1$
$B_1 = deposita \cdot B_2$ *sincronizzazione su deposita*
$B_2 = \overline{estrae} \cdot B_1$
$C_1 = estrae \cdot C_2$
$C_2 = consuma \cdot C_1$

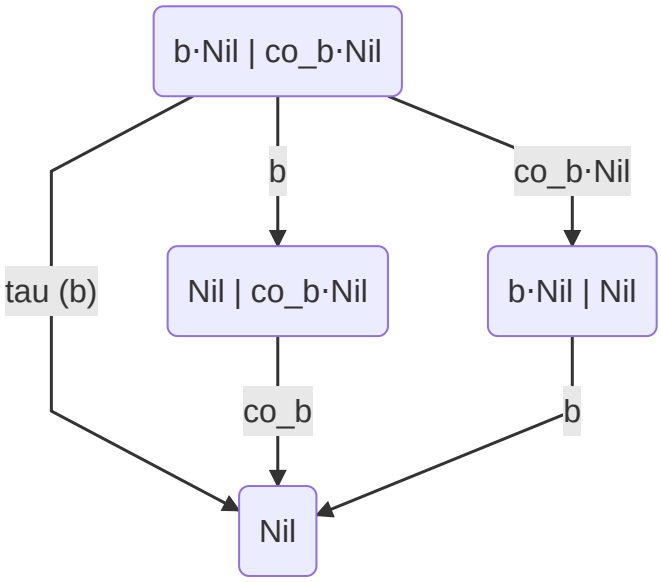
Abbiamo un sistema formato dalla composizione parallela di un processo produttore, un buffer e un consumatore. Vogliamo che questi processi interagiscano tra loro e che l'operazione di deposito nel buffer del produttore e quella di estrazione del consumatore dal buffer siano azioni interne al sistema *(il produttore non può depositare da un'altra parte e il consumatore non può consumare da un'altra parte)*.

Il sistema di transizioni associato a tale sistema è il seguente



Notiamo che la modellazione del sistema CCS viene fatta in maniera *sequenziale non deterministica*; cioè è come se ci fosse un unico processore che, in presenza di due processi paralleli indipendenti esegue prima uno e poi l'altro in ordine casuale.

NOTA: le azioni/coazioni potevano solamente sincronizzarsi a causa della restrizione. Consideriamo il processo $P = b \cdot Nil \mid \overline{b} \cdot Nil$: il suo LTS è



Questo accade perché senza la restrizione il processo P è libero di eseguire le azioni b e $\overline{b}$ con chiunque, anche con l'ambiente. Se immaginiamo l'azione b come la lettura da un buffer, applicare una restrizione su tale azione significa "forzare" il sistema a leggere **solo** dal buffer interno al sistema stesso e quindi a fare una sincronizzazione interna.

Non applicare alcuna restrizione invece lascia la scelta del buffer libera al sistema che di conseguenza può leggere dal suo buffer interno *(ed eseguire una sincronizzazione interna tau)* ma può anche leggere da un buffer dell'ambiente eseguendo, in ordine casuale, azione e coazione. Qui è ancora più chiara la modellazione sequenziale non deterministica. Vedremo esempi di vera concorrenza più avanti con le reti di Petri.

SPECIFICHE E IMPLEMENTAZIONI

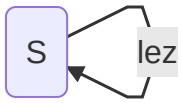
Col CCS possiamo anche fornire delle specifiche e delle implementazioni, ma vogliamo vedere come è possibile stabilire se una certa implementazione rispetta una specifica S.

Supponiamo $S = lez \cdot S$
Una possibile implementazione è il processo $Lucia\ Pomello$ che fa una lezione, mette una monetina nella macchinetta del caffè e si prende un caffè per poi ricominciare.
$LP = lez \cdot \overline{coin} \cdot \text{caffe} \cdot LP$

Possiamo vedere ora una possibile implementazione di S come $(LP \mid M) \setminus \{coin, \text{caffe}\}$ dove M è la macchinetta del caffè.

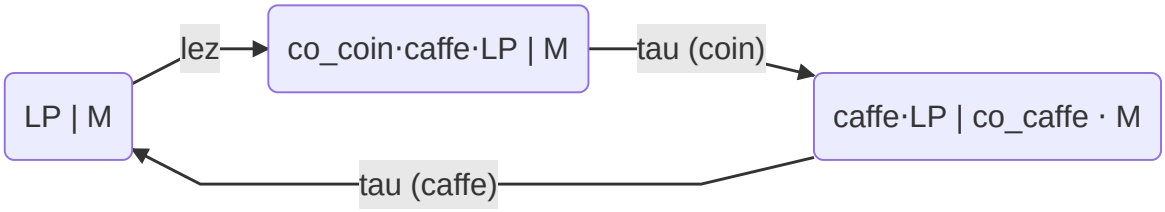
Vediamo anche come è fatta la macchinetta:
$M = coin \cdot \overline{\text{caffe}} \cdot M$

Vediamo ora il LTS della specifica:



In questo caso il sistema eroga in continuazione lezioni con l'ambiente.

E il LTS della specifica:



Notiamo che viene erogata la lezione con l'ambiente e poi succede *qualcosa* che l'ambiente però non vede; è come se gli studenti non vedessero la professoressa che prende il caffè ma solo il continuo flusso di lezioni.

EQUIVALENZA RISPETTO ALLE TRACCE

Abbiamo bisogno di una nozione che ci permetta di dire se l'implementazione soddisfa la specifica o meno o se due implementazioni soddisfano la stessa specifica.
Abbiamo bisogno di una **relazione di equivalenza** su P_{CCS} che astragga dagli stati, ovvero che consideri solo le azioni. In particolare vorremmo che questa relazione consideri solo le azioni col sistema ed astragga quindi dalle tau $\mathcal{T}$. Vogliamo inoltre astrarre dal non determinismo.

Vogliamo quindi che la nostra relazione sia:

- D'equivalenza
- Astragga dagli stati
- Astragga dalle tau $\mathcal{T}$
- Astragga dal non determinismo

Una relazione $R \subseteq P_{CCS} \times P_{CCS}$ di equivalenza è una **congruenza** sse $\forall\, p, q \in P_{CCS} \land \forall\, C[\cdot]$ se $p\ R\ q$ allora $C[p]\ R\ C[q]$.

Con $C[\cdot]$ contesto CCS, ovvero un processo in cui c'è una sorta di *variabile*.
Ad esempio $(LP \mid \cdot) \setminus \{coin,\ \text{caffe}\}$ è un contesto.

La prima idea di una relazione di questo tipo è andare a osservare i LTS dei due sistemi indagati e osservare se siano o meno isomorfi. Questa però è una restrizione troppo forte.

Un altro modo è osservare che vengano eseguite le stesse sequenze di interazione con l'ambiente. Questa nozione viene detta ***equivalenza rispetto alle tracce*** ed è definita come:
$p \in P_{CCS}\ Tracce(p) = \{w \in Act^* \mid \exists\, p' \in P_{CCS} \land p \to^w p'\}$

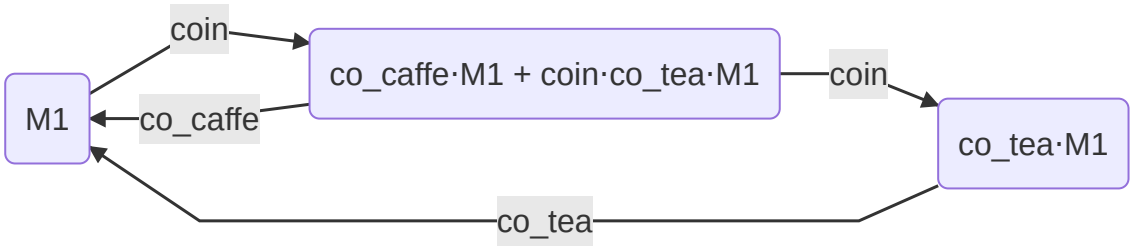
Diremo quindi che p_1 è equivalente rispetto alle tracce a p_2, scritto come $p_1 \sim^T p_2$ sse $Tracce(p_1) = Tracce(p_2)$.

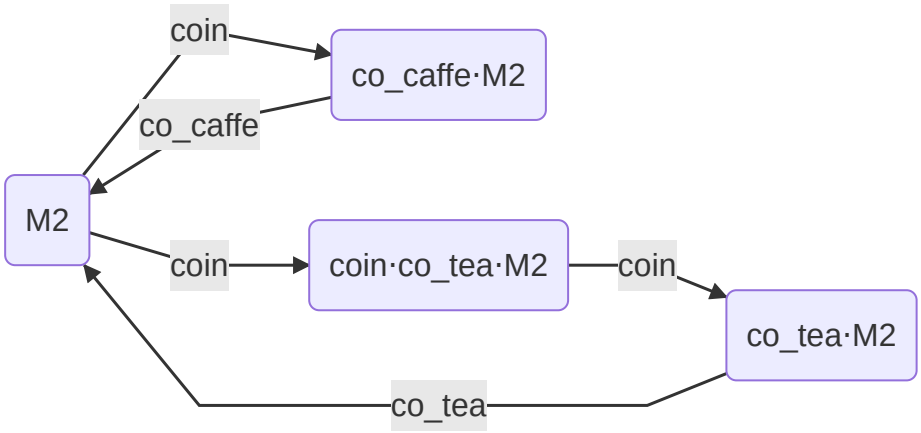
Consideriamo ora il sistema $(LP \mid M) \setminus \{coin, \text{caffe}\}$

Implementiamo M con $M_1 = coin(\overline{\text{caffe}} \cdot M_1 + coin \cdot \overline{tea} \cdot M_1)$
Ovvero la macchinetta M_1 può erogare anche il tè, che però costa due monete.

Prendiamo ora anche $M_2 = coin \cdot \overline{\text{caffe}} \cdot M_2 + coin \cdot coin \cdot \overline{tea} \cdot M_2$

Ci domandiamo ora se $M_1 \sim^T\ ?\ M_2$





Calcoliamo quindi le tracce della due macchinette:
$Tracce(M_1) = \{\epsilon,\ coin,\ coin \cdot \overline{\text{caffe}},\ coin \cdot coin,\ coin \cdot coin \cdot \overline{tea}\}$
$Tracce(M_2) = \{\epsilon,\ coin,\ coin \cdot \overline{\text{caffe}},\ coin \cdot coin,\ coin \cdot coin \cdot \overline{tea}\}$

Osserviamo che le due macchinette sono equivalenti rispetto alle tracce, ovvero $M_1 \sim^T M_2$.

Notiamo però che se vogliamo il caffè con la macchinetta M_1 possiamo sempre farlo avendo una scelta, con la macchinetta M_2 invece c'è un non determinismo che non dipende da noi e quindi, se vogliamo il caffè, il sistema potrebbe andare in deadlock scegliendo lo stato relativo al tè.
Quindi l'equivalenza rispetto alle tracce non è un equivalenza all'osservazione.

BISIMULAZIONE FORTE

Una delle equivalenze all'osservazione è la bisimulazione, che implica che due processi equivalenti devono *simularsi* a vicenda. Formalmente:
$R \subseteq P_{CCS} \times P_{CCS}$ è una **bisimulazione *forte*** se $\forall p, q \in P_{CCS} \ : \ p \ R \ q$ vale che:

- $\forall \alpha \in Act$ se $p \rightarrow^{\alpha} p'$ allora $\exists q' \in P_{CCS} \ : \ q \rightarrow^{\alpha} q' \wedge p' \ R \ q'$
  Cioè se possiamo fare un'azione in un processo, dobbiamo poterla fare anche nell'altro, ottenendo sempre una coppia di processi bisimili.
- $\forall \alpha \in Act$ se $q \rightarrow^{\alpha} q'$ allora $\exists p' \in P_{CCS} \ : \ p \rightarrow^{\alpha} p' \wedge p' \ R \ q'$

Quindi due processi p e q sono fortemente bisimili $p \sim^{BIS} q$ sse $\exists R \subseteq P_{CCS} \times P_{CCS}$ che è una bisimulazione forte tale per cui $p \ R \ q$ *(ne basta una, possono essercene di più)*.

Notiamo che $p \sim^{BIS} q \implies p \sim^{T} q$, ma non vale il viceversa.

La bisimulazione forte è una **congruenza rispetto a tutti gli operatori CCS**, questo significa che, avendo $p, q \in P_{CCS}$ con $p \sim^{BIS} q$, allora $\forall \alpha \in Act \wedge \forall r \in P_{CCS}$

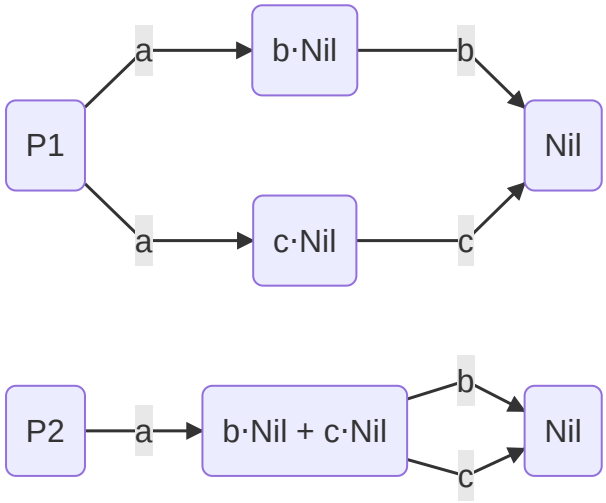
- $\alpha \cdot p \sim^{BIS} \alpha \cdot q$
- $p + r \sim^{BIS} q + r \wedge r + p \sim^{BIS} r + q$
- $p \mid r \sim^{BIS} q \mid r \wedge r \mid p \ \sim^{BIS} r \mid q$
- $p[f] \sim^{BIS} q[f] \, \forall f : Act \rightarrow Act$
- $p \setminus L \sim^{BIS} q \setminus L$

Possiamo anche dire che $p + Nil \sim^{BIS} p$ e che $p \mid Nil \sim^{BIS} p$.

Queste regole ci garantiscono sostituendo un processo da un sistema complesso con uno bisimile, il sistema di partenza rimane bisimile al sistema modificato.

ESEMPI

Consideriamo i processi $P_1 = a \cdot b \cdot Nil + a \cdot c \cdot Nil$ e $P_2 = a \cdot (b \cdot Nil + c \cdot Nil)$.
Costruiamo i loro LTS.



Osserviamo subito che i due processi sono equivalenti rispetto alle tracce, difatti
$Tracce(P_1) = \{\epsilon,\ a,\ a \cdot b,\ a \cdot c\}$
$Tracce(P_2) = \{\epsilon,\ a,\ a \cdot b,\ a \cdot c\}$

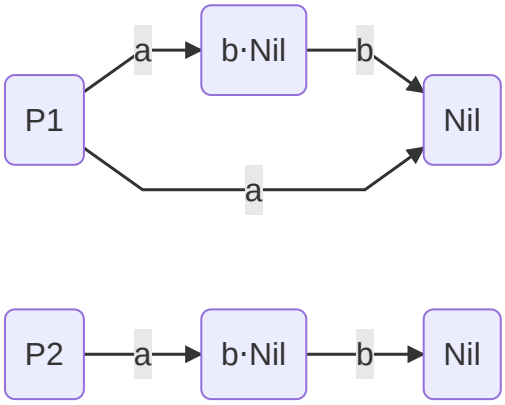
Vediamo se i due processi sono anche bisimili:
Da P_1 possiamo fare a e possiamo farlo anche da P_2; dobbiamo però chiederci se gli stati di arrivo sono in relazione di bisimulazione.
Gli stati interessati sono $b \cdot Nil$ e $b \cdot Nil + c \cdot Nil$: dal primo possiamo fare b, che è fattibile anche dal secondo, ma dal secondo possiamo fare c, che non è fattibile dal primo *(la bisimulazione richiede che entrambi gli stati siano simili tra loro)*, dunque i due processi ***non*** sono bisimili. Discorso analogo valeva se avessimo scelto come primo stato $c \cdot Nil$ che non può eseguire b.

Formalmente:
$p_1 \rightarrow^a b \cdot Nil$
$p_2 \rightarrow^a b \cdot Nil + c \cdot Nil$

E $b \cdot Nil \nsim^{BIS} b \cdot Nil + c \cdot Nil$

Notiamo che vale anche $b \cdot Nil \sim^{T} b \cdot Nil + c \cdot Nil$

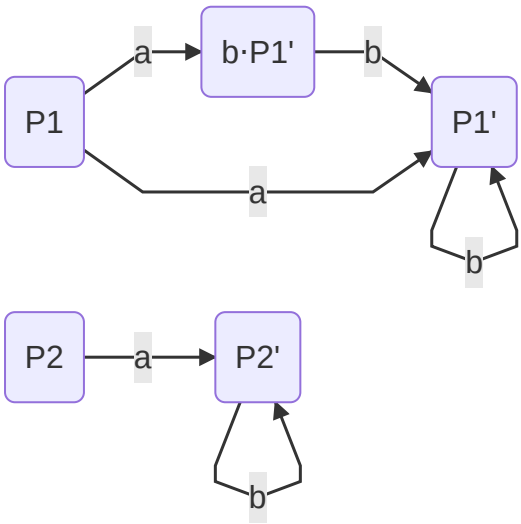
$P_1 = a \cdot b \cdot Nil + a \cdot Nil$ $\quad\quad$ $P_2 = a \cdot b \cdot Nil$



Anche questi due processi sono equivalenti rispetto alle tracce, difatti
$Tracce(P_1) = \{\epsilon,\ a,\ a \cdot b\}$
$Tracce(P_2) = \{\epsilon,\ a,\ a \cdot b\}$

Tuttavia non sono bisimili, poiché P_1 può eseguire a arrivando nello stato Nil che non può più eseguire alcuna azione, che non è bisimile allo stato $b \cdot Nil$ nel quale arriva P_2 dopo aver eseguito a poiché da quello stato si può eseguire b.

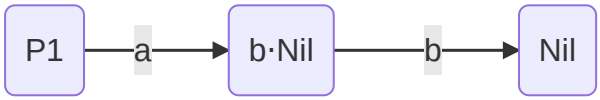
$P_1 = a \cdot b \cdot P_1' + a \cdot P_1'$ $\quad\quad$ $P_1' = b \cdot P_1'$
$P_2 = a \cdot P_2'$ $\quad\quad$ $P_2' = b \cdot P_2'$



Notiamo che i due processi sono equivalenti rispetto alle tracce, entrambi possono fare ab^*.
Si tratta inoltre di due processi bisimili:
Lo stato P_1 è bisimile allo stato P_2, mentre gli stati $b \cdot P_1'$ e P_1' sono entrambi bisimili allo stato P_2' poiché tutti e tre possono eseguire un numero arbitrario di volte b.

NOTA: osserviamo che nel processo P_1 abbiamo del non determinismo, mentre in P_2 no.

$P_1 = a \cdot b \cdot Nil$ $\quad\quad$ $P_2 = a \cdot \mathcal{T} \cdot b \cdot Nil$

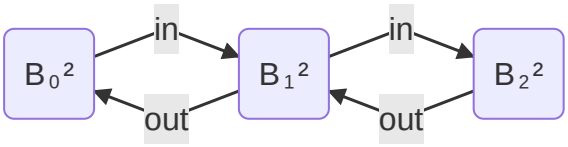


Osserviamo che questi due processi non sono nemmeno equivalenti rispetto alle tracce e quindi non sono nemmeno bisimili. Noi però vorremmo astrarre rispetto alle $\mathcal{T}$ e considerare solo le interazioni con l'ambiente indipendentemente dalle sincronizzazioni tra le componenti interne del sistema; non ci interessa com'è implementato il sistema, ma come esso interagisce col sistema.

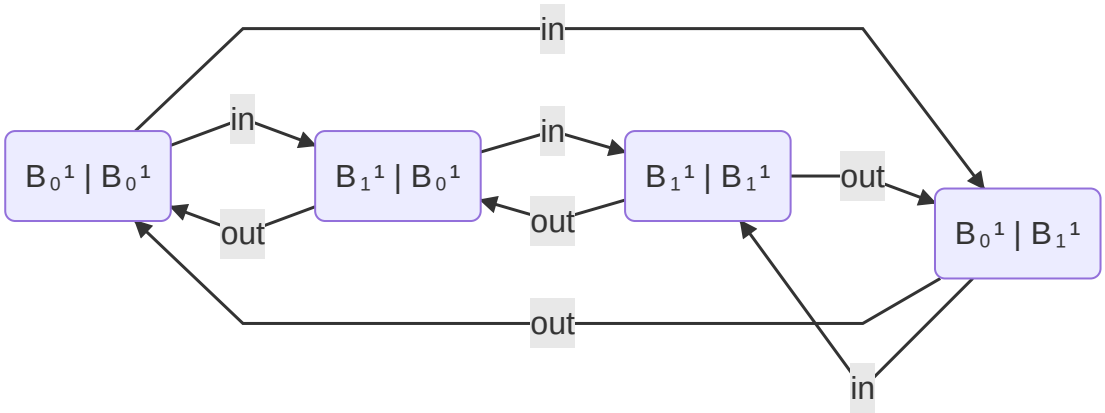
Questo esempio ci mostra come questa relazione sia troppo forte.

Un buffer a capacità uno è fatto in questo modo:
$B_0^1 = in \cdot B_1^1$ $\quad\quad$ $B_1^1 = \overline{out} \cdot B_0^1$

Quindi un buffer a capacità due sarà fatto come segue:
$B_0^2 = in \cdot B_1^2$ $\quad\quad$ $B_1^2 = \overline{out} \cdot B_0^2 + in \cdot B_2^2$ $\quad\quad$ $B_2^2 = \overline{out} \cdot B_1^2$



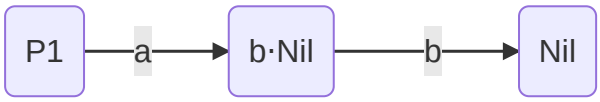
Implementiamo ora il buffer a capacità 2 usando due buffer a capacità uno in parallelo:
$B_0^1 \mid B_0^1$



Notiamo che gli stati B_0^2 e $B_0^1 \mid B_0^1$ sono bisimili. Lo sono anche B_1^2 con $B_1^1 \mid B_0^1$ e $B_0^1 \mid B_1^1$. Infine, anche gli stati B_2^2 e $B_1^1 \mid B_1^1$ sono bisimili.
I due processi sono quindi bisimili.

BISIMULAZIONE DEBOLE

Consideriamo i processi $p_1 = a \cdot b \cdot Nil$ e $p_2 = a \cdot \mathcal{T} \cdot b \cdot Nil$



Osserviamo subito che $p_1 \nsim^T p_2$, quindi a maggior ragione $p_1 \nsim^{BIS} p_2$.

Notiamo però che nel momento in cui un osservatore esterno osserva il comportamento dei due processi, non percepisce la sincronizzazione interna, ma solo le azioni con l'ambiente. Abbiamo quindi bisogno di una transizione debole $\Rightarrow^{\alpha}$ con $\alpha \in Act$, generalizzata alle sequenze come $\Rightarrow^{w}$ con $w \in Act^*$ che astragga dalle sincronizzazioni interne.
Con questa transizione possiamo dire che i due processi sono equivalenti rispetto alle tracce debolmente $p_1 \approx^T p_2$.

Con tale transizione debole potremmo anche definire la ***bisimulazione debole*** e dire che $p_1 \approx^{BIS} p_2$.

TRANSIZIONE DEBOLE

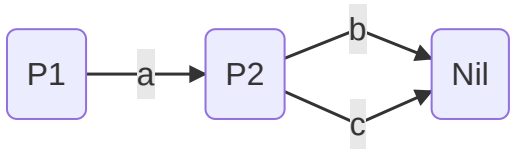
Siano $p,\ p' \in P_{CCS}$ e $\alpha \in Act = A \cup \overline{A} \cup \{\mathcal{T}\}$
$p \approx^T p'$ sse

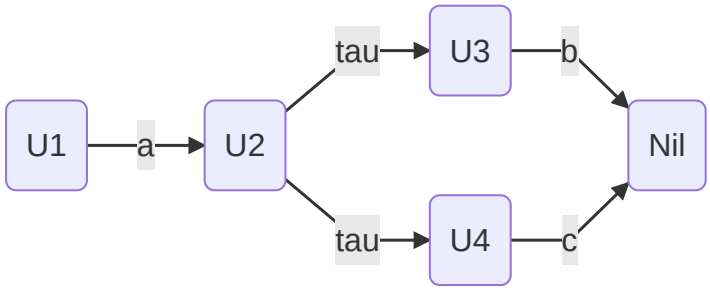
- $p \rightarrow^{\mathcal{T}^*} \rightarrow^{\alpha} \rightarrow^{\mathcal{T}^*} p'$ se $\alpha \neq \mathcal{T}$

  Ovvero se α è un'azione diversa da una sincronizzazione interna, p può eseguire un numero arbitrario di sincronizzazioni interne dopo le quali deve eseguire l'azione α e infine può ancora eseguire un numero arbitrario di sincronizzazioni interne e arrivare in p'.
- $p \rightarrow^{\mathcal{T}^*} p'$ se $\alpha = \mathcal{T}$

  Ovvero se α è una sincronizzazione interna, p può eseguire un numero arbitrario di sincronizzazioni interne per arrivare in p', anche zero.

Vediamo un esempio:
$p_1 = a \cdot (b \cdot Nil + c \cdot Nil) \qquad u_1 = a \cdot (\mathcal{T} \cdot b \cdot Nil + \mathcal{T} \cdot c \cdot Nil)$





Notiamo che $p_1 \nsim^T u_1$, quindi anche $p_1 \nsim^{BIS} u_1$.
Dal punto delle tracce deboli però abbiamo che $p_1 \approx^T u_1$.

Tuttavia, p_1 e u_1 hanno una '*diversa possibilità di generare deadlock*'. Se l'osservatore volesse fare $\overline{a} \cdot \overline{b}$, con p_1 può sempre farlo, mentre con u_1 può farlo solo se il processo sceglie la $\mathcal{T}$ verso u_3, mentre se sceglie quella verso u_4 si genera un deadlock. Vogliamo che la bisimulazione debole permetta di astrarre dalle $\mathcal{T}$, ma che continui a distinguere rispetto alla possibilità di generare deadlock.

BISIMULAZIONE DEBOLE: DEFINIZIONE

$R \subseteq P_{CCS} \times P_{CCS}$ è una bisimulazione debole sse
$\forall p,\ q \in P_{CCS} : p\ R\ q$ vale che $\forall \alpha \in Act$

- se $p \rightarrow^{\alpha} p_1$ allora esiste $q \approx^{\alpha} q_1$ tale che $p_1\ R\ q_1$

  Ovvero se p può eseguire α fortemente, q deve poter eseguire α debolmente.
- se $q \rightarrow^{\alpha} q_1$ allora esiste $p \approx^{\alpha} p_1$ tale che $p_1\ R\ q_1$

  (viceversa)

Anche in questo caso due processi sono debolmente bisimili $p \approx^{BIS} q$ se esiste una relazione *(ne basta una)* di bisimulazione debole tra loro $p\ R\ q$.

PROPRIETÀ

Prima di vedere le proprietà della bisimulazione debole, vediamo le proprietà di una relazione di equivalenza $\equiv$ tra processi

Siano $p, q \in P_{CCS}$

- se $LTS(p)$ è isomorfo a $LTS(q)$ allora $p \equiv q$
- deve astrarre dagli stati *(considera solo le azioni)*
- se $p \equiv q$ allora $Tracce(p) = Tracce(q)$
- se $p \equiv q$ allora p e q devono avere la stessa possibilità di generare deadlock interagendo con l'ambiente
- $\equiv$ deve essere una congruenza rispetto agli operatori CCS
  Cioè deve essere possibile sostituire un sottoprocesso con un altro equivalente senza modificare il comportamento complessivo del sistema

La prima equivalenza che abbiamo visto è l'equivalenza rispetto alle tracce forte; essa è una congruenza rispetto agli operatori ed astrae dagli stati, ma non garantisce la preservazione della possibilità di generare deadlock.

Abbiamo poi visto la bisimulazione forte che permette di preservare la possibilità di generare deadlock, ma è una relazione troppo forte che non astrae dalle sincronizzazioni interne.

Abbiamo introdotto anche le tracce deboli, da cui è possibile ottenere un'equivalenza rispetto alle tracce debole che astrae dagli stati, è una congruenza ma non riesce a preservare la stessa probabilità di deadlock.

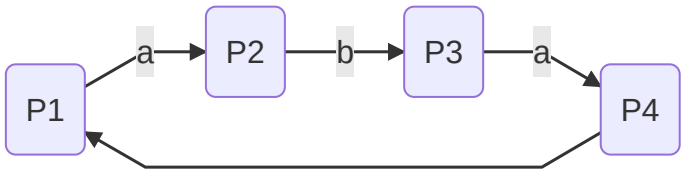
Siamo infine arrivati alla bisimulazione debole.

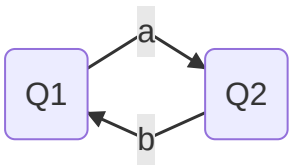
PROCESSI DETERMINISTICI ED EQUIVALENZE

Un processo $p \in P_{CCS}$ è deterministivo sse $\forall a \in Act$, se $p \to^a p' \wedge p \to^a p''$ allora $p' = p''$.

Se due processi p e q sono deterministici e $p \sim^T q$ *(*$p \approx^T q$*)* allora $p \sim^{BIS} q$ *(*$p \approx^{BIS}$*)*.

$p = a \cdot b \cdot a \cdot p$ $\qquad$ $q = a \cdot b \cdot q$





Notiamo che i due processi sono deterministici e sono anche $p \sim^T q$ *(e quindi anche* $p \approx^T q$*)*, di conseguenza sono anche bisimili.

GIOCO ATTACCANTE E DIFENSORE

Possiamo confrontare due processi p e q usando un ***gioco*** $G(p, q)$ con due giocatori:

- **Attaccante**: cerca di dimostrare $p \not\approx^{BIS} q$
- **Difensore**: cerca di dimostrare $p \approx^{BIS} q$

Il gioco ha **più partite**.
Un partita di un gioco $G(p, q)$ è una sequenza finita o infinita di **configurazioni** $(p_0, q_0), (p_1, q_1) \dots (p_i, q_i)$ con $(p, q) = (p_0, q_0)$. Una configurazione non è altro che una mossa eseguita su un processo che è poi eseguita anche sull'altro *(se possibile)*.

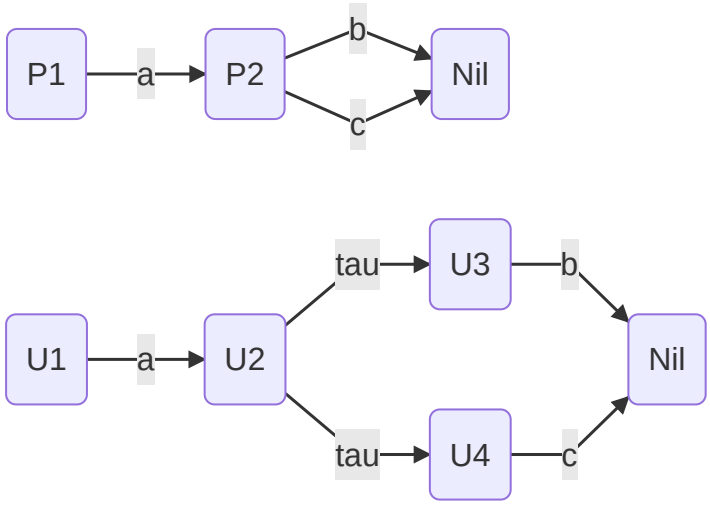
In ogni ***mano*** si passa dalla configurazione corrente (p_i, q_i) alla successiva (p_{i+1}, q_{i+1}) con le seguenti regole:

- L'attaccante inizia e sceglie uno dei due processi della configurazione corrente (p_i, q_i) e fa una mossa forte $\rightarrow^{\alpha}$ *(l'attaccante può scegliere un processo diverso ad ogni mano)*
- Il difensore deve eseguire, sull'altro processo, la stessa mossa fatta dall'attaccante ma debole $\Rightarrow^{\alpha}$

La coppia di processi così ottenuta (p_{i+1}, q_{i+1}) diventa la nuova configurazione.

Se un giocatore non può muovere, l'altro vince. Se la partita è infinita vince il difensore.

Diverse partite possono tuttavia concludersi con vincitori diversi:

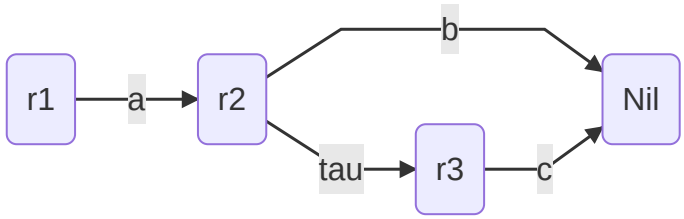


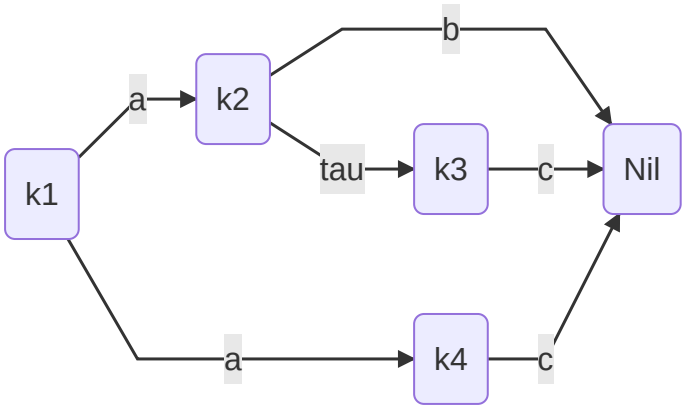
1. L'attaccante fa $p_1 \rightarrow^{a} p2$, il difensore risponde con $u_1 \Rightarrow^{a} u_2$.
   L'attaccante fa $p_2 \rightarrow^{b} Nil$, il difensore risponde con $u_2 \Rightarrow^{b} Nil$ *(fa $tau \cdot b$)*
   Il difensore vince.
2. L'attaccante fa $p_1 \rightarrow^{a} p2$, il difensore risponde con $u_1 \Rightarrow^{a} u_2$.
   L'attaccante fa $u_2 \rightarrow^{\mathcal{T}} u_3$, il difensore risponde con $p_2 \Rightarrow^{\mathcal{T}^*} p_2$ *(sta fermo)*
   L'attaccante fa $p_2 \rightarrow^{c} Nil$, il difensore non può rispondere.
   L'attaccante vince.

Tuttavia, per ogni gioco, solo uno dei de giocatori può vincere **ogni** partita usando una ***strategia vincente***, ovvero una sorta di insieme di regole che indicano di volta in volta la mossa da fare. Le regole dipendono dalla configurazione corrente.
Diremo che $p_1 \not\approx^{BIS} u_1$ se l'attaccante ha una strategia vincente.
Diremo che $p_1 \approx^{BIS} u_1$ se il difensore ha una strategia vincente.

ESEMPI

$r_1 = a \cdot (b \cdot Nil + \mathcal{T} \cdot c \cdot Nil)$ $\quad$ $k_1 = a \cdot (b \cdot Nil + \mathcal{T} \cdot c \cdot Nil) + a \cdot c \cdot Nil$



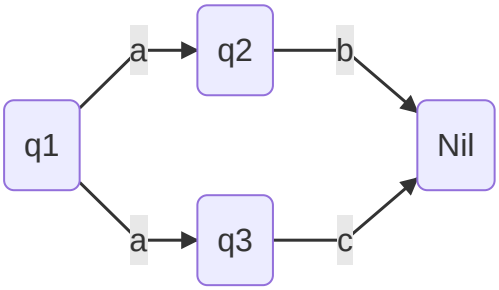


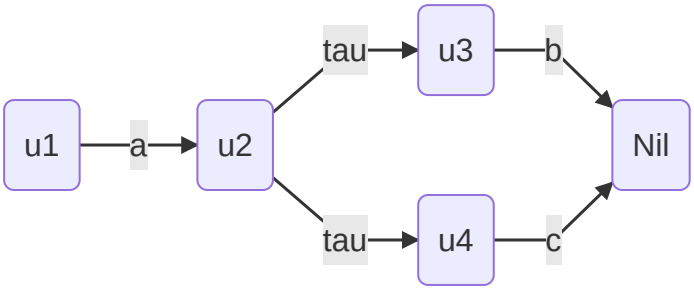
Il difensore ha una strategia vincente, ma per dimostrarla dobbiamo far vedere che riesce ad applicarla per ogni mossa dell'attaccante.
All'inizio l'attaccante può fare tre mosse:

1. A $r_1 \rightarrow^a r2$
    1. D $k_1 \Rightarrow^a k_2$
       I sottoprocessi ottenuti sono isomorfi, quindi sicuramente bisimili e quindi il difensore vince.
2. A $k_1 \rightarrow^a k_2$
    1. D $r_1 \Rightarrow^a r_2$
       Stesso caso di prima.
3. A $k_1 \rightarrow^a k_4$
    1. D $r_1 \Rightarrow^a r_3$
       Troviamo ancora dei sottoprocessi isomorfi, quindi il difensore vince.

Quindi $r_1 \approx^{BIS} k_1$

$q_1 = a \cdot b \cdot Nil + a \cdot c \cdot Nil$ $\quad$ $u_1 = a \cdot (\mathcal{T} \cdot b \cdot Nil + \mathcal{T} \cdot c \cdot Nil)$





L'attaccante ha una strategia vincente, ma per dimostrarla bisogna mostrare che è valida per ogni mossa del difensore.

L'attaccante inizia con: A $u_1 \rightarrow^a u_2$
Il difensore può fare:

1. D $q_1 \Rightarrow^a q_2$
    1. A $u_2 \rightarrow^{\mathcal{T}} u_4$ $\quad$ D $q_2 \Rightarrow^{\mathcal{T}^{*}0} q_2$
    2. A $u_4 \rightarrow^c Nil$ $\quad$ Il difensore non può rispondere.
2. D $q_1 \Rightarrow^a q_3$
    1. A $u_2 \rightarrow^{\mathcal{T}} u_3$ $\quad$ D $q_2 \Rightarrow^{\mathcal{T}^{*}0} q_2$
    2. A $u_3 \rightarrow^c Nil$ $\quad$ Il difensore non può rispondere.

Quindi $q_1 \not\approx^{BIS} u_1$